W3School Swift 教程

wizardforcel

Published with GitBook



目錄

W3School Swift 教程

作者：W3School

来源：Swift 教程

Swift 简介



Swift 是一种支持多编程范式和编译式的开源编程语言,苹果于2014年WWDC（苹果开发者大会）发布，用于开发 iOS，OS X 和 watchOS应用程序。

Swift 结合了 C 和 Objective-C 的优点并且不受 C 兼容性的限制。

Swift在 Mac OS 和 iOS 平台可以和Object-C使用相同的运行环境。这意味着Swift程序可以运行于目前已存在的平台之上，包含 iOS 6 和 OS X 10.8 都可以运行Swift的程序。

更重要的, Swift 和 Obj-C 的代码可并存于单一程序内, 这种延伸就如同 C 和 C++ 的关系一样。

2015年6月8日，苹果于WWDC 2015上宣布，Swift将开放源代码，包括编译器和标准库。

谁适合阅读本教程？

本教程适合想从事移动端(iphone)开发或 OS X 应用的编程人员，如果之前有编程基础更好。

本教程所有实例基于 Xcode7.1（Swift 2.x 的语法格式） 开发测试。

第一个 Swift 程序

第一个 Swift 程序当然从输出 "Hello, World!" 开始，代码如下所示：

```
/* 我的第一个 Swift 程序 */
var myString = "Hello, World!"

print(myString)
```

Swift 环境搭建

Swift是一门开源的编程语言，该语言用于开发OS X和iOS应用程序。

在正式开发应用程序前，我们需要搭建Swift开发环境，以便更好友好的使用各种开发工具和语言进行快速应用开发。由于Swift开发环境需要在OS X系统中运行，因此其环境的搭建将不同于Windows环境，下面就一起来学习一下swift开发环境的搭建方法。

成功搭建swift开发环境的前题：

1. 必须拥有一台苹果电脑。因为集成开发环境XCode只能运行在OS X系统上。
2. 电脑系统必须在OS 10.9.3及以上。
3. 电脑必须安装Xcode集成开发环境。

Swift 开发工具Xcode下载

Swift 开发工具官网地址：https://developer.apple.com/xcode/download/。

Swift 开发工具百度软件中心下载（国内比较快）：http://rj.baidu.com/soft/detail/40233.html

Swift 源代码下载：https://swift.org/download/#latest-development-snapshots

下载完成后，双击下载的 dmg 文件安装，安装完成后我们将 Xcode 图标踢移动到应用文件夹。

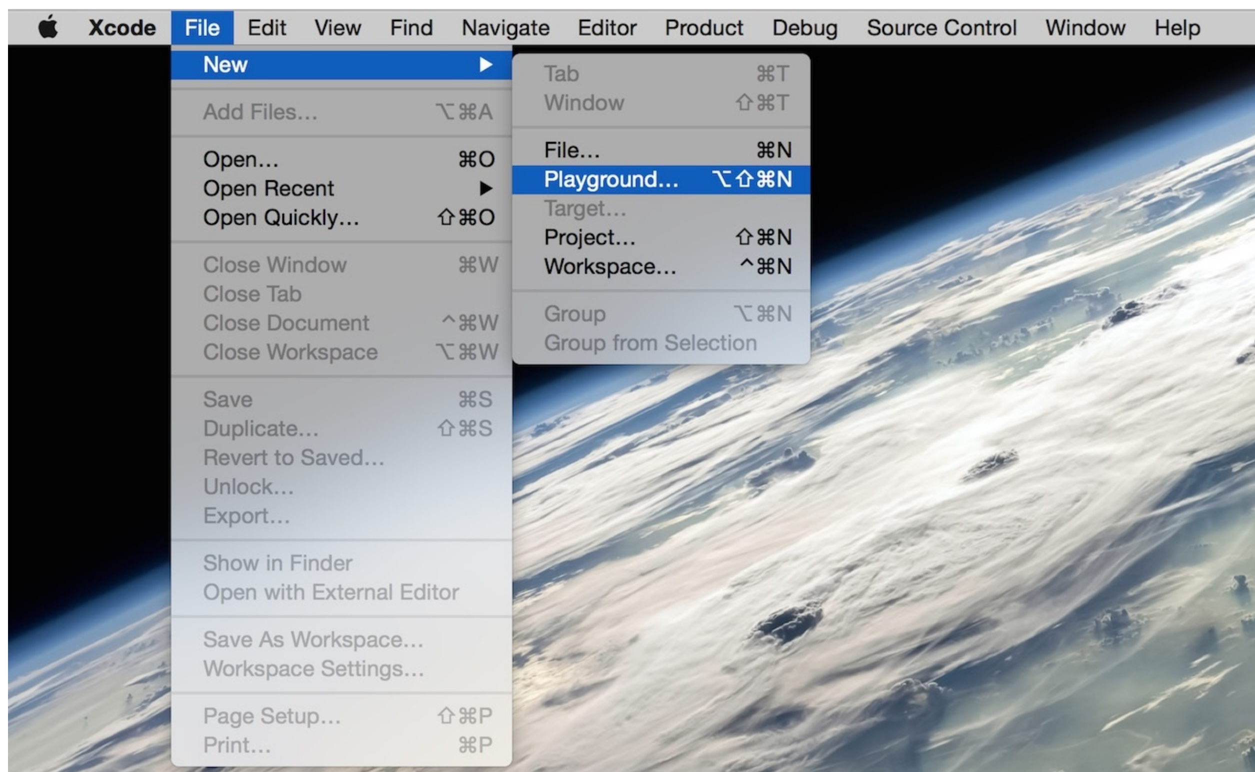


你也可以在 App Store 中搜索 xcode 安装，如下图所示：

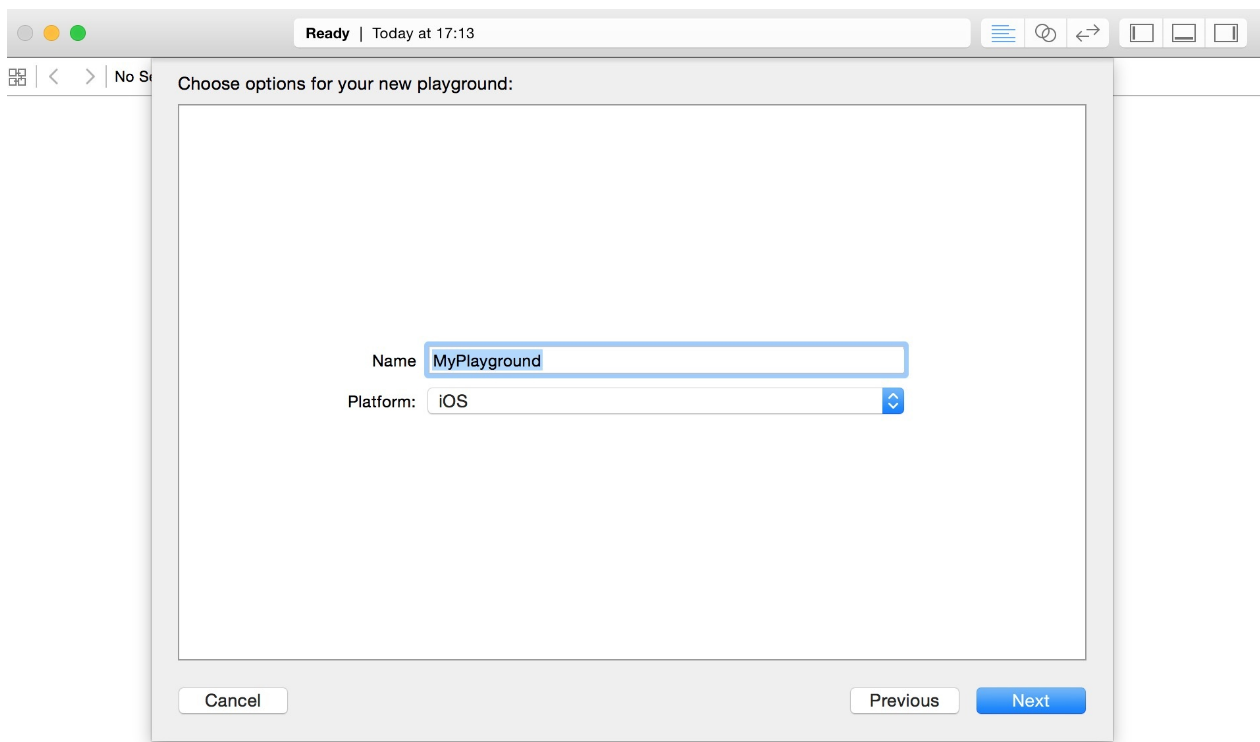
第一个 Swift 程序

Xcode 安装完成后，我们就可以开始编写 Swift 代码了。

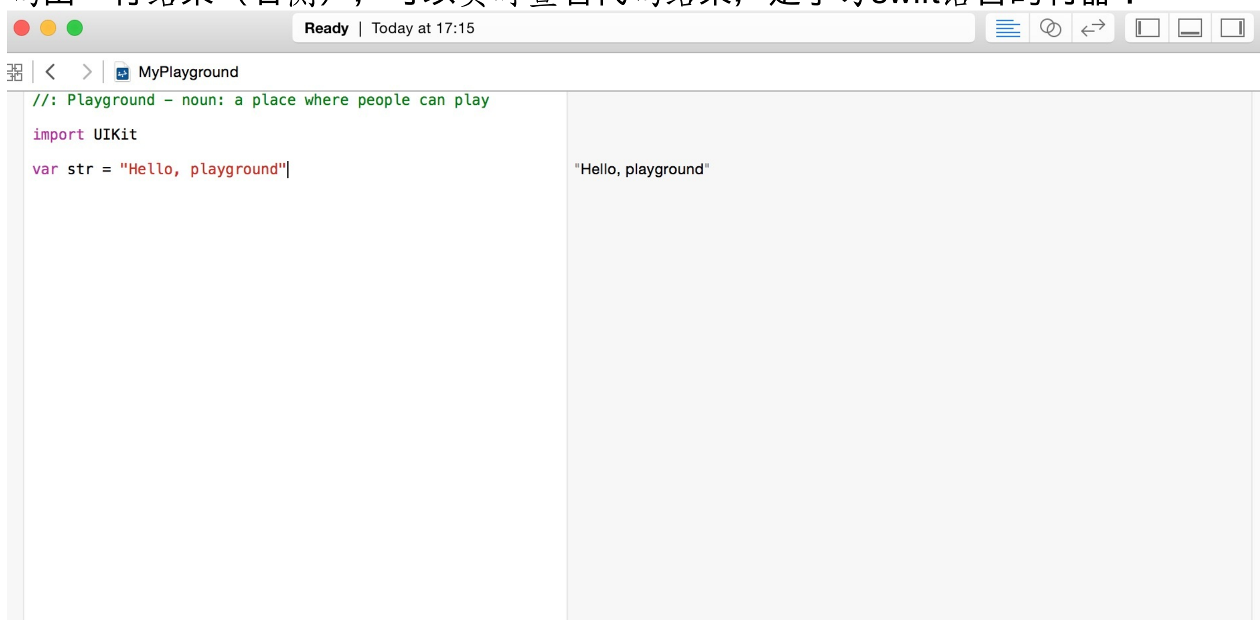
接下来我们在应用文件夹打开 Xcode，打开后在屏幕顶部选择 File => New => Playground。



接着 为 playground 设置一个名字并选择 iOS 平台。



Swift 的 playground 就像是一个可交互的文档，它是用来练手学swift的，写一句代码出一行结果（右侧），可以实时查看代码结果，是学习swift语言的利器！



以下是 Swift Playground 窗口默认的代码：

```
import UIKit

var str = "Hello, playground"
```

如果你想创建 OS x 程序，需要导入 Cocoa 包 **import Cocoa** 代码如下所示：

```
import Cocoa

var str = "Hello, playground"
```

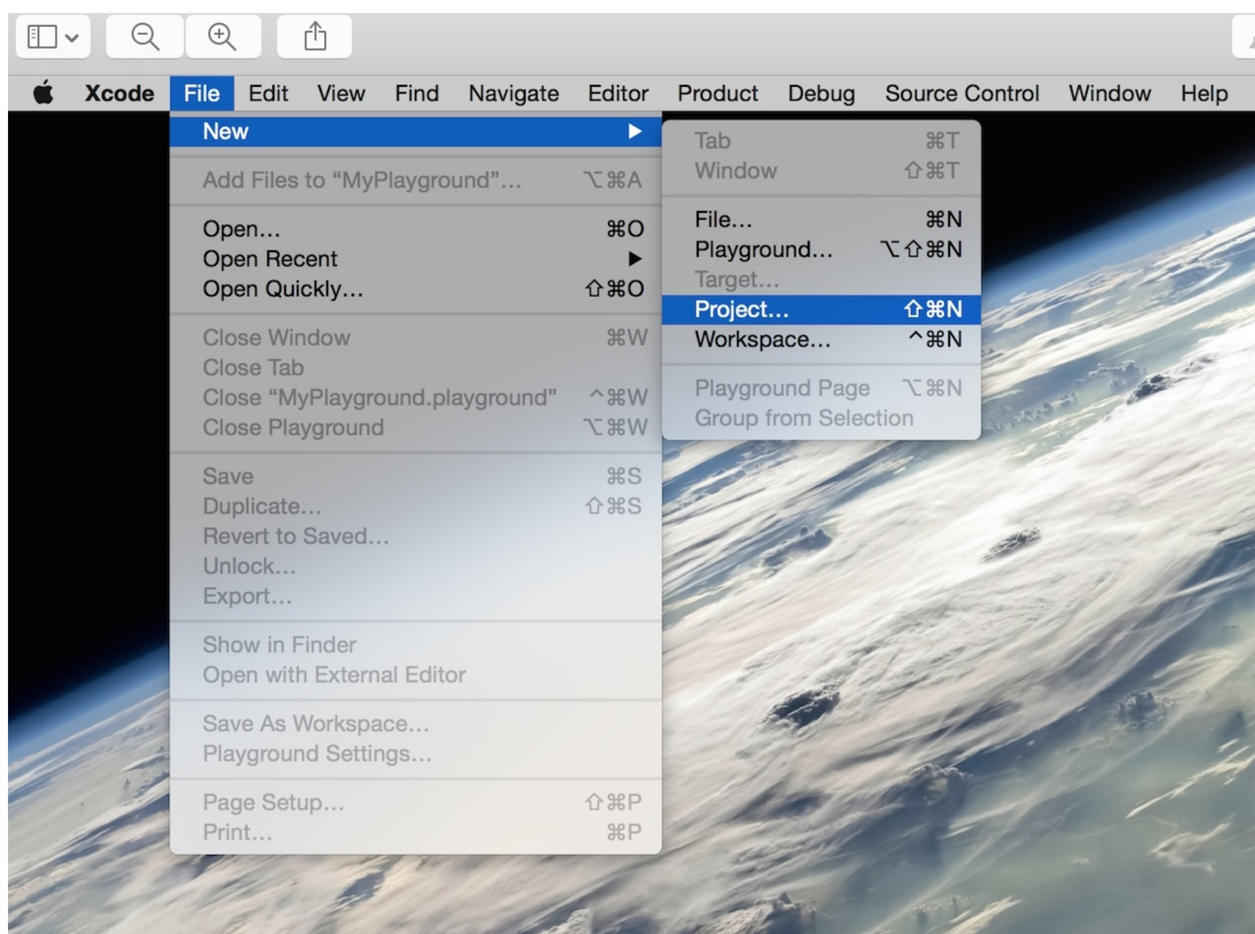
以上程序载入后，会在Playground 窗口右侧显示程序执行结果：

```
Hello, playground
```

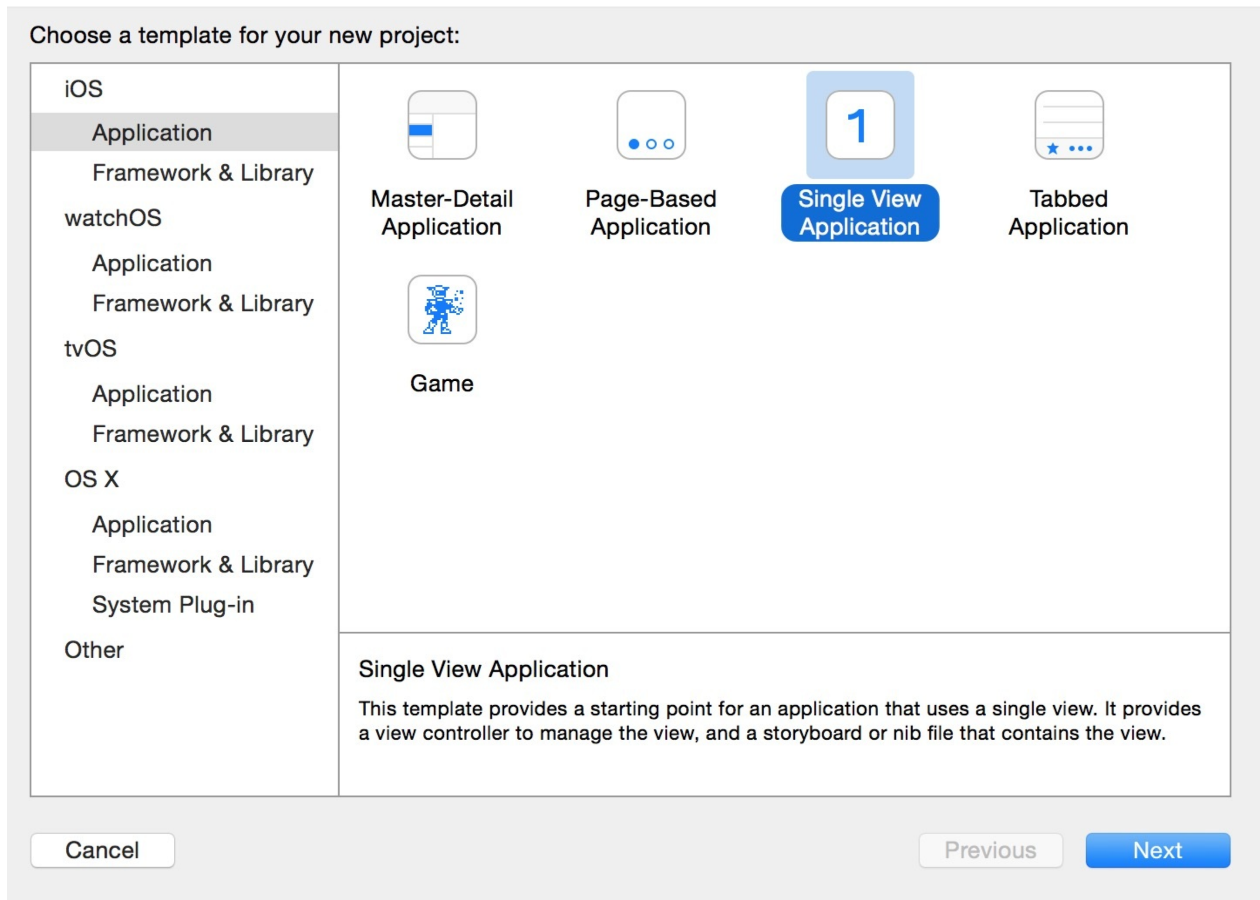
至此，你已经完成了第一个 Swift 程序的学习，恭喜你入门了。

创建第一个项目

1、打开 xcode 工具，选择 File => New => Project



2、我们选择一个"Single View Application"，并点击"next"，创建一个简单示例app应用。



3、接着我们输入项目名称（ProductName),公司名称（Organization Name),公司标识前缀名（Organization identifier) 还要选择开发语言(Language),选择设备（Devices)。

其中Language有两个选项：Objective-c和swift，因为我们是学习swift当然选择swift项了。 点击"Next"下一步。

Choose options for your new project:

Product Name: HelloWorld

Organization Name: 菜鸟教程

Organization Identifier: com.runoob

Bundle Identifier: com.runoob.HelloWorld

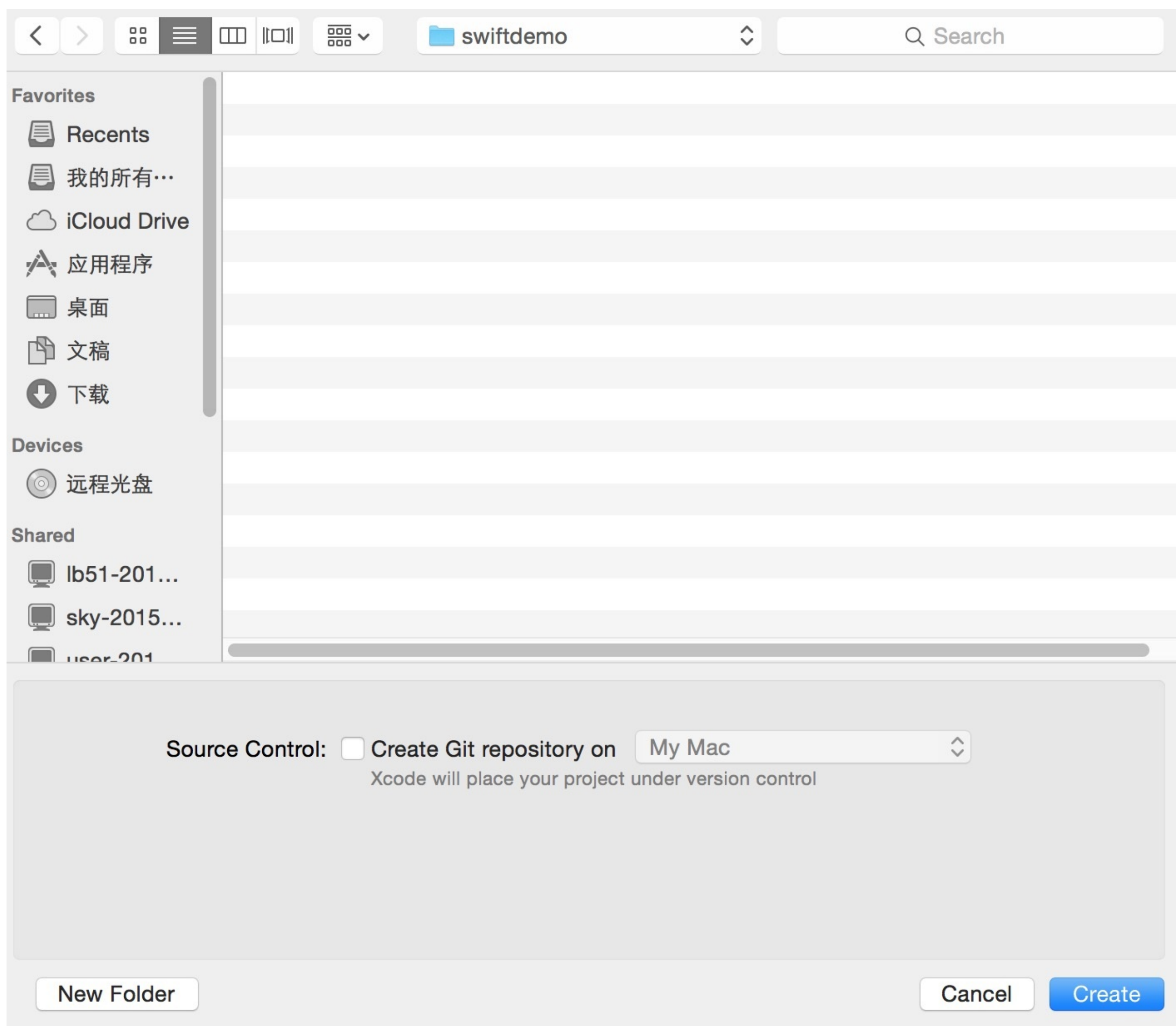
Language: Swift

Devices: iPhone

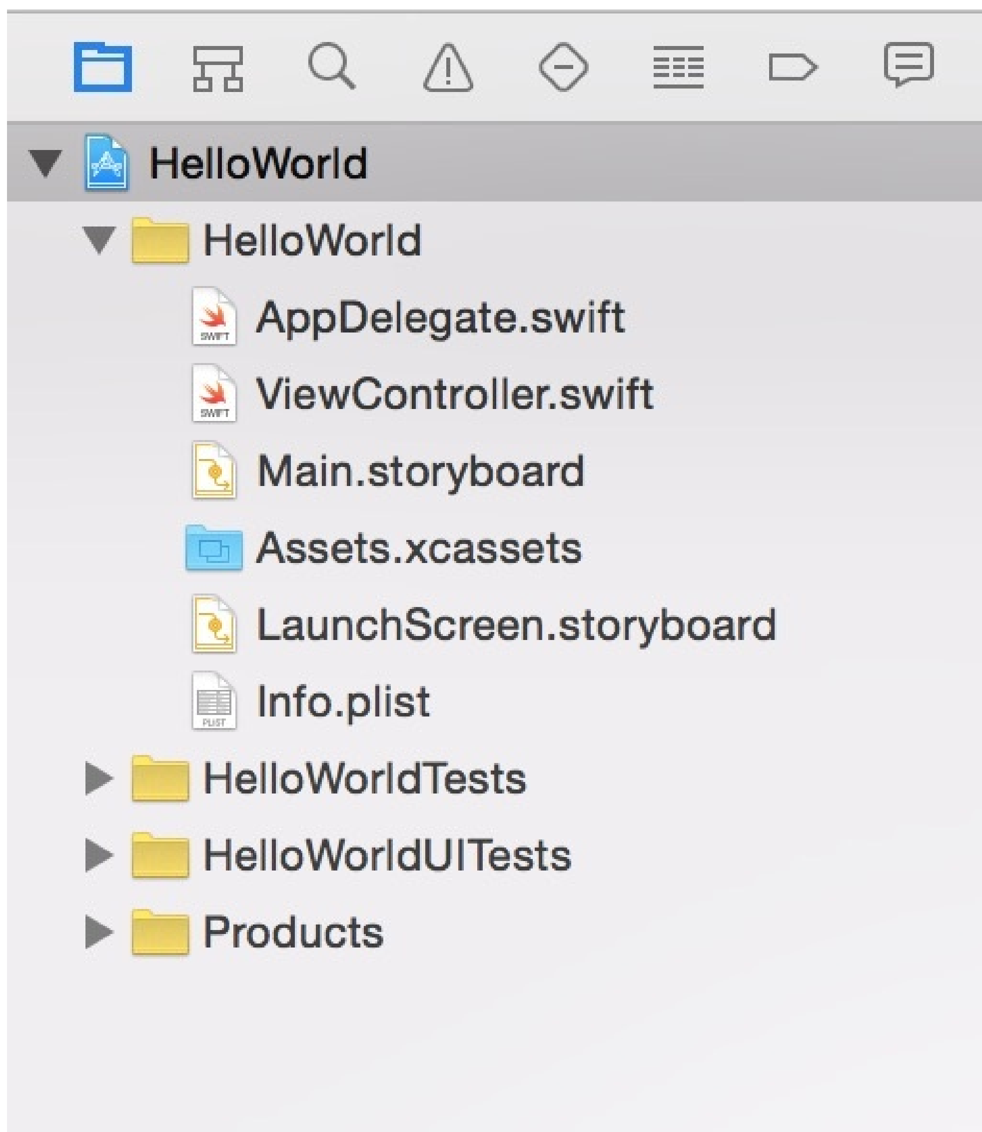
- [ ] Use Core Data
- [x] Include Unit Tests
- [x] Include UI Tests

Cancel　Previous　Next

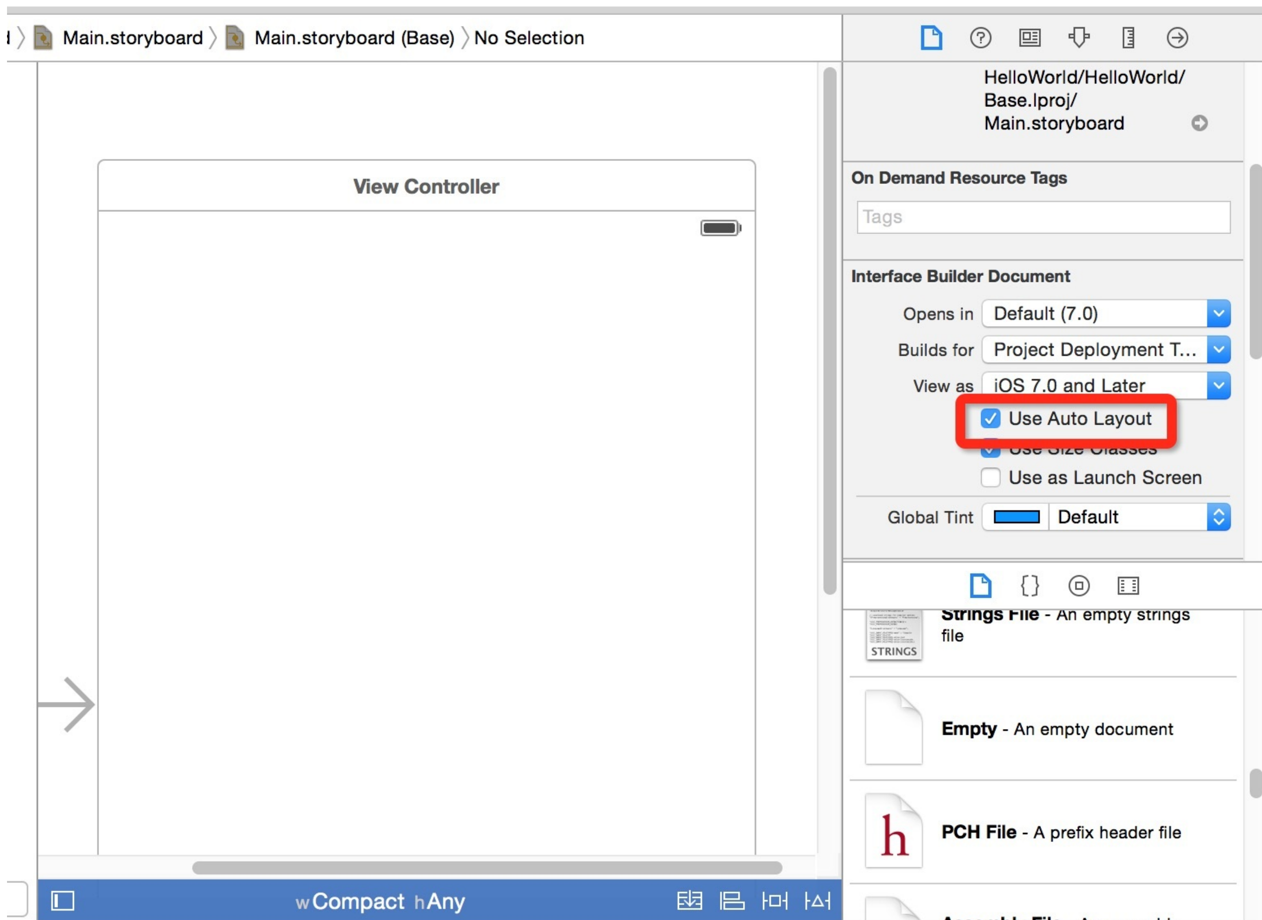
4、选择存放的目录，如果要使用Git源代码管理，将勾上Source Control的create git repository on My Mac. 点击create创建项目。



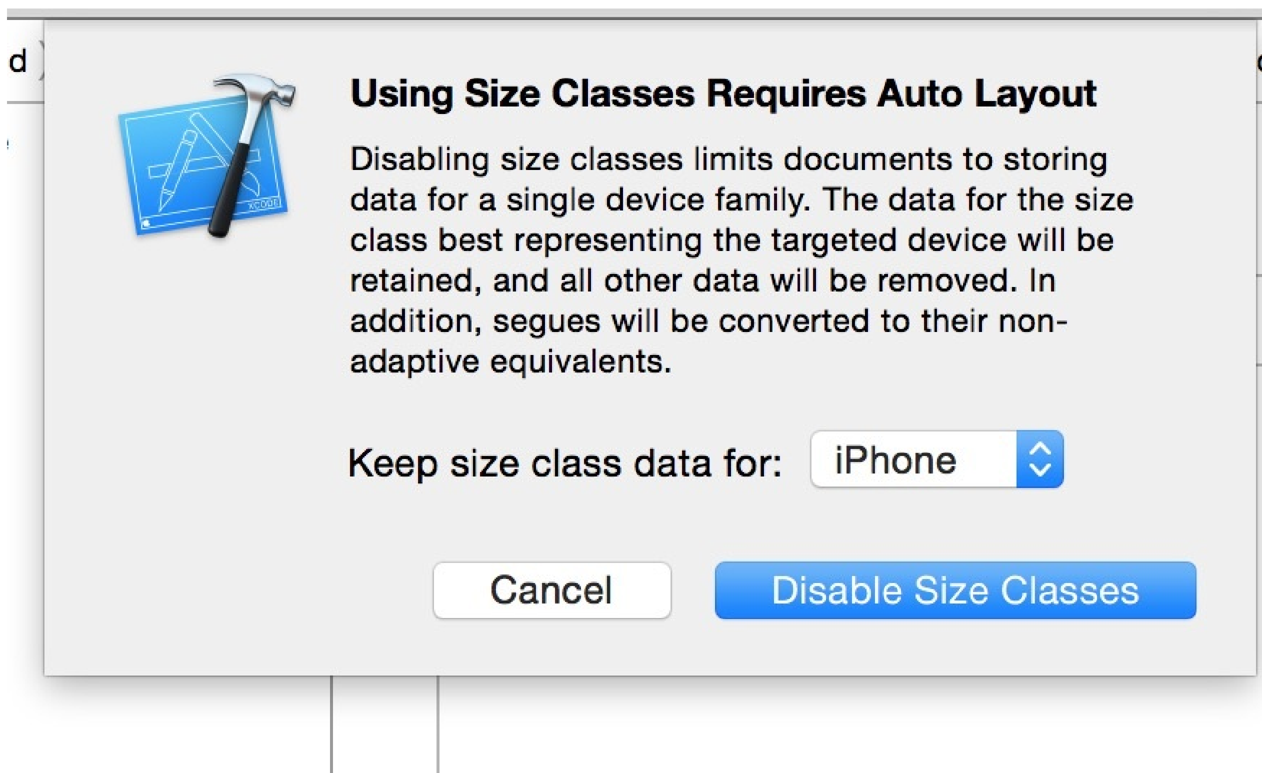
5、项目创建后，默认生成了一个示例文件，可以看到swift将oc中的h和m文件合并成了一个文件（即swift后缀名文件). Main.storyboard相当于xib文件，有比xib更多的功能。



6、打开main.storyboard,默认看到一个简单的空白的应用界面，大小为平板界面大小。 如果开发都只需要开发兼容iphone手机的app,那么可以把Use Auto Layout的勾去掉（默认为勾上）。



7、弹出了一个对话框，让我们选择界面尺寸，iPhone或都 iPad。我们选择iPhone的尺寸。

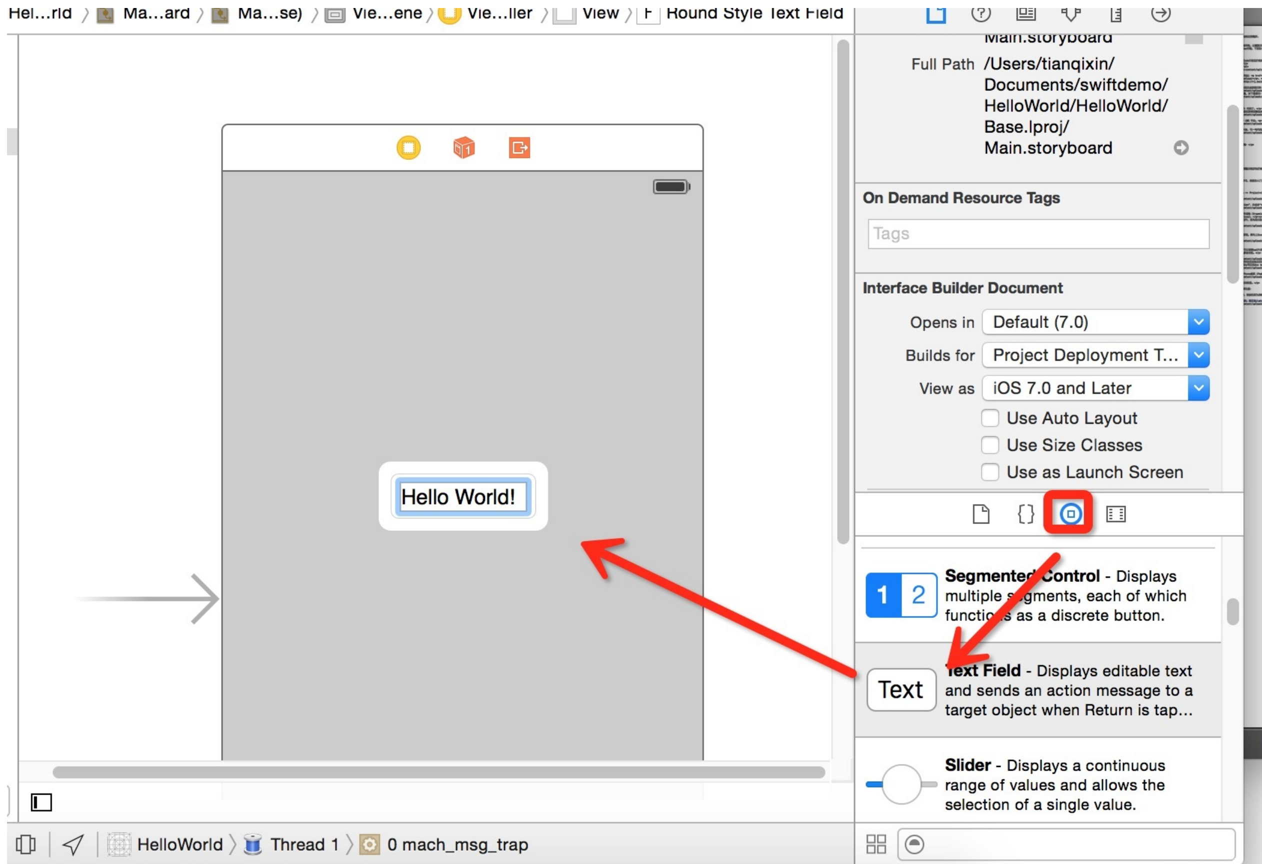


8、可以看到，界面大小变为了手机iphone的宽度和高度。

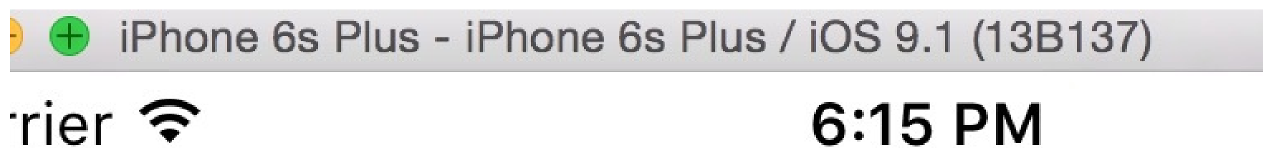
大家可以记住界面相关的尺寸，方便以后布局计算位置：

iPhone或iTouch的宽为320像素，高为480像素，状态栏高为20像素，toobar高为44像素，tabbar高为49像素，导航栏高为44像素。

9.我们为界面添加点内容，在右下方找到Text控件，将它拖入storyboard上，并双击写入文本"Hello World!"。



运行一下模拟器（command+R 快捷键或在菜单栏中选择 Product => Run)。



至此，我们的第一个Swift项目就完成了。

Swift 基本语法

在上一章节中我们已经讲到如何创建 Swift 语言的 "Hello, World!" 程序。现在我们来复习下。

如果创建的是 OS X playground 需要引入 Cocoa ：

```
import Cocoa

/* 我的第一个 Swift 程序 */
var myString = "Hello, World!"

print(myString)
```

如果我们想创建 iOS playground 则需要引入 UIKit :

```
import UIKit
var myString = "Hello, World!"
print(myString)
```

执行以上程序，输出结果为：

```
Hello, World!
```

以上代码即为 Swift 程序的基本结构，接下来我们来详细说明结构的组成部分。

Swift 引入

我们可以使用 **import** 语句来引入任何的 Objective-C 框架（或 C 库）到 Swift 程序中。例如 **import cocoa** 语句导入了使用了 Cocoa 库和API，我们可以在 Swift 程序中使用他们。

Cocoa 本身由 Objective-C 语言写成，Objective-C 又是 C 语言的严格超集，所以在 Swift 应用中我们可以很简单的混入 C 语言代码，甚至是 C++ 代码。

Swift 标记

Swift 程序由多种标记组成，标记可以是单词，标识符，常量，字符串或符号。例如以下 Swift 程序由三种标记组成：

```
print("test!")
标记是：单词、符号
print
(
   "test!"
)
```

注释

Swift的注释与C语言极其相似，单行注释以两个反斜线开头：

```
//这是一行注释
```

多行注释以/开始，以/结束:

```
/* 这也是一条注释,
但跨越多行 */
```

与 C 语言的多行注释有所不同的是，Swift 的多行注释可以嵌套在其他多行注释内部。写法是在一个多行注释块内插入另一个多行注释。第二个注释块封闭时，后面仍然接着第一个注释块：

```
/* 这是第一个多行注释的开头

/* 这是嵌套的第二个多行注释 */

这是第一个多行注释的结尾 */
```

多行注释的嵌套是你可以更快捷方便的注释代码块，即使代码块中已经有了注释。

分号

与其它语言不同的是，Swift不要求在每行语句的结尾使用分号(;)，但当你在同一行书写多条语句时，必须用分号隔开：

```
import Cocoa
/* 我的第一个 Swift 程序 */
var myString = "Hello, World!"; print(myString)
```

标识符

标识符就是给变量、常量、方法、函数、枚举、结构体、类、协议等指定的名字。构成标识符的字母均有一定的规范，Swift语言中标识符的命名规则如下：

- 区分大小写，Myname与myname是两个不同的标识符；
- 标识符首字符可以以下划线（_）或者字母开始，但不能是数字；
- 标识符中其他字符可以是下划线（_）、字母或数字。

例如： userName、User_Name、_sys_val、身高等为合法的标识符，而2mail、room#和class为非法的标识符。

注意:Swift中的字母采用的是Unicode编码[1]。Unicode叫做统一编码制，它包含了亚洲文字编码，如中文、日文、韩文等字符，甚至是我们在聊天工具中使用的表情符号

如果一定要使用关键字作为标识符，可以在关键字前后添加重音符号（`），例如：

关键字

关键字是类似于标识符的保留字符序列，除非用重音符号（`）将其括起来，否则不能用作标识符。关键字是对编译器具有特殊意义的预定义保留标识符。常见的关键字有以下4种。

与声明有关的关键字

class	deinit	enum	extension
func	import	init	internal
let	operator	private	protocol
public	static	struct	subscript
typealias	var		

与语句有关的关键字

break	case	continue	default
do	else	fallthrough	for
if	in	return	switch
where	while		

表达式和类型关键字

as	dynamicType	false	is
nil	self	Self	super
true	*COLUMN*	*FILE*	*FUNCTION*
LINE			

在特定上下文中使用的关键字

associativity	convenience	dynamic	didSet
final	get	infix	inout
lazy	left	mutating	none
nonmutating	optional	override	postfix
precedence	prefix	Protocol	required
right	set	Type	unowned
weak	willSet		

Swift 空格

Swift语言并不是像C/C++，Java那样完全忽视空格，Swift对空格的使用有一定的要求，但是又不像Python对缩进的要求那么严格。

在Swift中，运算符不能直接跟在变量或常量的后面。例如下面的代码会报错：

```
let a= 1 + 2
```

错误信息是：

```
error: prefix/postfix '=' is reserved
```

意思大概是等号直接跟在前面或后面这种用法是保留的。

下面的代码还是会报错（继续注意空格）：

```
let a = 1+ 2
```

错误信息是：

```
error: consecutive statements on a line must be separated by ';'
```

这是因为Swift认为到1+这个语句就结束了，2就是下一个语句了。

只有这样写才不会报错：

```
let a = 1 + 2;  // 编码规范推荐使用这种写法
let b = 3+4 // 这样也是OK的
```

Swift 字面量

所谓字面量，就是指像特定的数字，字符串或者是布尔值这样，能够直接了当地指出自己的类型并为变量进行赋值的值。比如在下面：

```
42                   // 整型字面量
3.14159              // 浮点型字面量
"Hello, world!"      // 字符串型字面量
true                 // 布尔型字面量
```

Swift 数据类型

在我们使用任何程序语言编程时，需要使用各种数据类似来存储不同的信息。

变量的数据类型决定了如何将代表这些值的位存储到计算机的内存中。在声明变量时也可指定它的数据类型。

所有变量都具有数据类型，以决定能够存储哪种数据。

内置数据类型

Swift 提供了非常丰富的数据类型，以下列出了常用了集中数据类型：

Int

一般来说，你不需要专门指定整数的长度。Swift 提供了一个特殊的整数类型 `Int` ，长度与当前平台的原生字长相同：

- 在32位平台上， `Int` 和 `Int32` 长度相同。
- 在64位平台上， `Int` 和 `Int64` 长度相同。

除非你需要特定长度的整数，一般来说使用 `Int` 就够了。这可以提高代码一致性和可复用性。即使是在32位平台上， `Int` 可以存储的整数范围也可以达到 `-2,147,483,648` ~ `2,147,483,647` ，大多数时候这已经足够大了。

UInt

Swift 也提供了一个特殊的无符号类型 `UInt` ，长度与当前平台的原生字长相同：

- 在32位平台上， `UInt` 和 `UInt32` 长度相同。
- 在64位平台上， `UInt` 和 `UInt64` 长度相同。

> 注意：
>
> 尽量不要使用 `UInt` ，除非你真的需要存储一个和当前平台原生字长相同的无符号整数。除了这种情况，最好使用 `Int` ，即使你要存储的值已知是非负的。统一使用 `Int` 可以提高代码的可复用性，避免不同类型数字之间的转换，并且匹配数字的类型推断，请参考类型安全和类型推断。

浮点数

浮点数是有小数部分的数字，比如 `3.14159` ， `0.1` 和 `-273.15` 。

浮点类型比整数类型表示的范围更大，可以存储比 `Int` 类型更大或者更小的数字。Swift 提供了两种有符号浮点数类型：

- `Double` 表示64位浮点数。当你需要存储很大或者很高精度的浮点数时请使用此类型。
- `Float` 表示32位浮点数。精度要求不高的话可以使用此类型。

> 注意：
>
> `Double` 精确度很高，至少有15位数字，而 `Float` 最少只有6位数字。选择哪个类型取决于你的代码需要处理的值的范围。

布尔值

Swift 有一个基本的布尔（Boolean）类型，叫做Bool。布尔值指逻辑上的值，因为它们只能是真或者假。Swift 有两个布尔常量，true和false。

字符串

字符串是字符的序列集合，例如：

```
"Hello, World!"
```

字符

字符指的是单个字母，例如：

```
"C"
```

可选类型

使用可选类型（optionals）来处理值可能缺失的情况。可选类型表示有值或没有值。

数值范围

下表显示了不同变量类型内存的存储空间，及变量类型的最大最小值：

类型	大小（字节）	区间值
Int8	1 字节	-127 到 127
UInt8	1 字节	0 到 255
Int32	4 字节	-2147483648 到 2147483647
UInt32	4 字节	0 到 4294967295
Int64	8 字节	-9223372036854775808 到 9223372036854775807
UInt64	8 字节	0 到 18446744073709551615
Float	4 字节	1.2E-38 到 3.4E+38 (~6 digits)
Double	8 字节	2.3E-308 到 1.7E+308 (~15 digits)

类型别名

类型别名对当前的类型定义了另一个名字，类型别名通过使用 typealias 关键字来定义。语法格式如下：

```
typealias newname = type
```

例如以下定义了 Int 的类型别名为 Feet：

```
typealias Feet = Int
```

现在，我们可以通过别名来定义变量：

```
import Cocoa

typealias Feet = Int
var distance: Feet = 100
print(distance)
```

我们使用 playground 执行以上程序，输出结果为：

```
100
```

类型安全

Swift 是一个类型安全（type safe）的语言。

由于 Swift 是类型安全的，所以它会在编译你的代码时进行类型检查（type checks），并把不匹配的类型标记为错误。这可以让你在开发的时候尽早发现并修复错误。

```
import Cocoa

var varA = 42
varA = "This is hello"
print(varA)
```

以上程序，会在 Xcode 中报错：

```
error: cannot assign value of type 'String' to type 'Int'
varA = "This is hello"
```

意思为不能将 'String' 字符串赋值给 'Int' 变量。

类型推断

当你要处理不同类型的值时，类型检查可以帮你避免错误。然而，这并不是说你每次声明常量和变量的时候都需要显式指定类型。

如果你没有显式指定类型，Swift 会使用类型推断（type inference）来选择合适的类型。

例如，如果你给一个新常量赋值42并且没有标明类型，Swift 可以推断出常量类型是Int，因为你给它赋的初始值看起来像一个整数：

```
let meaningOfLife = 42
// meaningOfLife 会被推测为 Int 类型
```

同理，如果你没有给浮点字面量标明类型，Swift 会推断你想要的是Double：

```
let pi = 3.14159
// pi 会被推测为 Double 类型
```

当推断浮点数的类型时，Swift 总是会选择Double而不是Float。

如果表达式中同时出现了整数和浮点数，会被推断为Double类型：

```
let anotherPi = 3 + 0.14159
// anotherPi 会被推测为 Double 类型
```

原始值3没有显式声明类型，而表达式中出现了一个浮点字面量，所以表达式会被推断为Double类型。

实例

```
import Cocoa

// varA 会被推测为 Int 类型
var varA = 42
print(varA)

// varB 会被推测为 Double 类型
var varB = 3.14159
print(varB)

// varC 也会被推测为 Double 类型
var varC = 3 + 0.14159
print(varC)
```

执行以上代码，输出结果为：

```
42
3.14159
3.14159
```

Swift 变量

变量是一种使用方便的占位符，用于引用计算机内存地址。

Swift 每个变量都指定了特定的类型，该类型决定了变量占用内存的大小，不同的数据类型也决定可存储值的范围。

上一章节我们已经为大家介绍了基本的数据类型，包括整形Int、浮点数Double和Float、布尔类型Bool以及字符串类型String。此外，Swift还提供了其他更强大数据类型， Optional, Array, Dictionary, Struct, 和 Class 等。

接下来我们将为大家介绍如何在 Swift 程序中声明和使用变量。

变量声明

变量声明意思是告诉编译器在内存中的哪个位置上为变量创建多大的存储空间。

在使用变量前，你需要使用 **var** 关键字声明它，如下所示：

```
var variableName = <initial value>
```

以下是一个 Swift 程序中变量声明的简单实例：

```
import Cocoa

var varA = 42
print(varA)

var varB:Float

varB = 3.14159
print(varB)
```

以上程序执行结果为：

```
42
3.14159
```

变量命名

变量名可以由字母，数字和下划线组成。

变量名需要以字母或下划线开始。

Swift 是一个区分大小写的语言，所以字母大写与小写是不一样的。

变量名也可以使用简单的 Unicode 字符，如下实例：

```
import Cocoa

var _var = "Hello, Swift!"
print(_var)

var 你好 = "你好世界"
var 菜鸟教程 = "www.runoob.com"
print(你好)
print(菜鸟教程)
```

以上程序执行结果为：

```
Hello, Swift!
你好世界
www.runoob.com
```

变量输出

变量和常量可以使用 **print**（swift 2 将 print 替换了 println） 函数来输出。

在字符串中可以使用括号与反斜线来插入变量，如下实例：

```
import Cocoa

var name = "菜鸟教程"
var site = "http://www.runoob.com"

print("\(name)的官网地址为：\(site)")
```

以上程序执行结果为：

```
菜鸟教程的官网地址为：http://www.runoob.com
```

Swift 可选(Optionals)类型

Swift 的可选（Optional）类型，用于处理值缺失的情况。可选表示"那儿有一个值，并且它等于 x "或者"那儿没有值"。

Swfit语言定义后缀？作为命名类型Optional的简写，换句话说，以下两种声明是相等的：

```
var optionalInteger: Int?
var optionalInteger: Optional<Int>
```

在这两种情况下，变量optionalInteger都是可选整数类型。注意，在类型和？之间没有空格。

Optional 是一个含有两种情况的枚举，None和Some(T)，用来表示可能有或可能没有值。任何类型都可以明确声明为（或者隐式转换）可选类型。当声明一个可选类型的时候，要确保用括号给？操作符一个合适的范围。例如，声明可选整数数组，应该写成(Int[])?；写成Int[]?会报错。

当你声明一个可选变量或者可选属性的时候没有提供初始值，它的值会默认为nil。

可选项遵照LogicValue协议，因此可以出现在布尔环境中。在这种情况下，如果可选类型T?包含类型为T的任何值（也就是说它的值是Optional.Some(T)），这个可选类型等于true，反之为false。

如果一个可选类型的实例包含一个值，你可以用后缀操作符 ！来访问这个值，如下所示：

```
optionalInteger = 42
optionalInteger! // 42
```

使用操作符！去获取值为nil的可选变量会有运行时错误。

你可以用可选链接和可选绑定选择性执行可选表达式上的操作。如果值为nil，任何操作都不会执行，也不会有运行报错。

让我们来详细看下以下实例来了解 Swift 中可选类型的应用:

```
import Cocoa

var myString:String? = nil

if myString != nil {
    print(myString)
}else{
    print("字符串为 nil")
}
```

以上程序执行结果为：

```
字符串为 nil
```

可选类型类似于Objective-C中指针的nil值，但是nil只对类(class)有用，而可选类型对所有的类型都可用，并且更安全。

强制解析

当你确定可选类型确实包含值之后，你可以在可选的名字后面加一个感叹号（!）来获取值。这个感叹号表示"我知道这个可选有值，请使用它。"这被称为可选值的强制解析（forced unwrapping）。

实例如下：

```
import Cocoa

var myString:String?

myString = "Hello, Swift!"

if myString != nil {
   print(myString)
}else{
   print("myString 值为 nil")
}
```

以上程序执行结果为：

```
Optional("Hello, Swift!")
```

强制解析可选值，使用感叹号（!）：

```
import Cocoa

var myString:String?

myString = "Hello, Swift!"

if myString != nil {
   // 强制解析
   print( myString! )
}else{
   print("myString 值为 nil")
}
```

以上程序执行结果为：

```
Hello, Swift!
```

> 注意：
>
> 使用 `!` 来获取一个不存在的可选值会导致运行时错误。使用 `!` 来强制解析值之前，一定要确定可选包含一个非 `nil` 的值。

自动解析

你可以在声明可选变量时使用感叹号（!）替换问号（?）。这样可选变量在使用时就不需要再加一个感叹号（!）来获取值，它会自动解析。

实例如下：

```
import Cocoa

var myString:String!

myString = "Hello, Swift!"

if myString != nil {
   print(myString)
}else{
   print("myString 值为 nil")
}
```

以上程序执行结果为：

```
Hello, Swift!
```

可选绑定

使用可选绑定（optional binding）来判断可选类型是否包含值，如果包含就把值赋给一个临时常量或者变量。可选绑定可以用在if和while语句中来对可选类型的值进行判断并把值赋给一个常量或者变量。

像下面这样在if语句中写一个可选绑定：

```
if let constantName = someOptional {
    statements
}
```

让我们来看下一个简单的可选绑定实例：

```
import Cocoa

var myString:String?

myString = "Hello, Swift!"

if let yourString = myString {
   print("你的字符串值为 - \(yourString)")
}else{
   print("你的字符串没有值")
}
```

以上程序执行结果为：

```
你的字符串值为 - Hello, Swift!
```

Swift 常量

常量一旦设定，在程序运行时就无法改变其值。

常量可以是任何的数据类型如：整型常量，浮点型常量，字符常量或字符串常量。同样也有枚举类型的常量：

常量类似于变量，区别在于常量的值一旦设定就不能改变，而变量的值可以随意更改。

常量声明

常量使用关键字 **let** 来声明，语法如下：

```
let constantName = <initial value>
```

以下是一个简单的 Swift 程序中使用常量的实例：

```
import Cocoa

let constA = 42
print(constA)
```

以上程序执行结果为：

```
42
```

类型标注

当你声明常量或者变量的时候可以加上类型标注（type annotation），说明常量或者变量中要存储的值的类型。如果要添加类型标注，需要在常量或者变量名后面加上一个冒号和空格，然后加上类型名称。

```
var constantName:<data type> = <optional initial value>
```

以下是一个简单是实例演示了 Swift 中常量使用类型标注。需要注意的是常量定义时必须初始值：

```
import Cocoa

let constA = 42
print(constA)

let constB:Float = 3.14159

print(constB)
```

以上程序执行结果为：

```
42
3.14159
```

常量命名

常量的命名可以由字母，数字和下划线组成。

常量需要以字母或下划线开始。

Swift 是一个区分大小写的语言，所以字母大写与小写是不一样的。

常量名也可以使用简单的 Unicode 字符，如下实例：

```
import Cocoa

let _const = "Hello, Swift!"
print(_const)

let 你好 = "你好世界"
print(你好)
```

以上程序执行结果为：

```
Hello, Swift!
你好世界
```

常量输出

变量和常量可以使用 print（swift 2 将 print 替换了 println） 函数来输出。

在字符串中可以使用括号与反斜线来插入常量，如下实例：

```
import Cocoa

let name = "菜鸟教程"
let site = "http://www.runoob.com"

print("\(name)的官网地址为：\(site)")
```

以上程序执行结果为：

```
菜鸟教程的官网地址为：http://www.runoob.com
```

Swift 字面量

所谓字面量，就是指像特定的数字，字符串或者是布尔值这样，能够直接了当地指出自己的类型并为变量进行赋值的值。比如在下面：

```
let aNumber = 3          //整型字面量
let aString = "Hello"    //字符串字面量
let aBool = true         //布尔值字面量
```

整型字面量

整型字面量可以是一个十进制，二进制，八进制或十六进制常量。 二进制前缀为 0b，八进制前缀为 0o，十六进制前缀为 0x，十进制没有前缀：

以下为一些整型字面量的实例：

```
let decimalInteger = 17           // 17 - 十进制表示
let binaryInteger = 0b10001       // 17 - 二进制表示
let octalInteger = 0o21           // 17 - 八进制表示
let hexadecimalInteger = 0x11     // 17 - 十六进制表示
```

浮点型字面量

浮点型字面量有整数部分，小数点，小数部分及指数部分。

除非特别指定，浮点型字面量的默认推导类型为 Swift 标准库类型中的 Double，表示64位浮点数。

浮点型字面量默认用十进制表示（无前缀），也可以用十六进制表示（加前缀 0x）。

> 十进制浮点型字面量由十进制数字串后跟小数部分或指数部分（或两者皆有）组成。十进制小数部分由小数点 . 后跟十进制数字串组成。指数部分由大写或小写字母 e 为前缀后跟十进制数字串组成，这串数字表示 e 之前的数量乘以 10 的几次方。例如：1.25e2 表示 1.25 × 10^2，也就是 125.0；同样，1.25e－2 表示 1.25 × 10^－2，也就是 0.0125。
>
> 十六进制浮点型字面量由前缀 0x 后跟可选的十六进制小数部分以及十六进制指数部分组成。十六进制小数部分由小数点后跟十六进制数字串组成。指数部分由大写或小写字母 p 为前缀后跟十进制数字串组成，这串数字表示 p 之前的数量乘以 2 的几次方。例如：0xFp2 表示 15 × 2^2，也就是 60；同样，0xFp-2 表示 15 × 2^-2，也就是 3.75。
>
> 负的浮点型字面量由一元运算符减号 - 和浮点型字面量组成，例如 -42.5。

浮点型字面量允许使用下划线 _ 来增强数字的可读性，下划线会被系统忽略，因此不会影响字面量的值。同样地，也可以在数字前加 0，并不会影响字面量的值。

以下为一些浮点型字面量的实例：

```
let decimalDouble = 12.1875       //十进制浮点型字面量
let exponentDouble = 1.21875e1    //十进制浮点型字面量
let hexadecimalDouble = 0xC.3p0   //十六进制浮点型字面量
```

字符串型字面量

字符串型字面量由被包在双引号中的一串字符组成，形式如下：

```
"characters"
```

字符串型字面量中不能包含未转义的双引号 （"）、未转义的反斜线（\）、回车符或换行符。

转移字符	含义
\0	空字符
\\	反斜线 \
\b	退格(BS) ，将当前位置移到前一列
\f	换页(FF)，将当前位置移到下页开头
\n	换行符
\r	回车符
\t	水平制表符
\v	垂直制表符
\'	单引号
\"	双引号
\000	1到3位八进制数所代表的任意字符
\xhh...	1到2位十六进制所代表的任意字符

以下为字符串字面量的简单实例：

```
import Cocoa

let stringL = "Hello\tWorld\n\n菜鸟教程官网：\'http://www.runoob.com\
print(stringL)
```

以上程序执行结果为：

```
Hello	World

菜鸟教程官网：'http://www.runoob.com'
```

布尔型字面量

布尔型字面量的默认类型是 Bool。

布尔值字面量有三个值，它们是 Swift 的保留关键字：

- **true** 表示真。
- **false** 表示假。

- **nil** 表示没有值。

Swift 运算符

运算符是一个符号，用于告诉编译器执行一个数学或逻辑运算。

Swift 提供了以下几种运算符：

- 算术运算符
- 比较运算符
- 逻辑运算符
- 位运算符
- 赋值运算符
- 区间运算符
- 其他运算符

本章节我们将为大家详细介绍算术运算符、关系运算符、逻辑运算符、位运算符、赋值运算符及其他运算符。

算术运算符

以下表格列出了 Swift 语言支持的算术运算符，其中变量 A 为 10，变量 B 为 20：

运算符	描述	实例
+	加号	A + B 结果为 30
−	减号	A − B 结果为 -10
*	乘号	A * B 结果为 200
/	除号	B / A 结果为 2
%	求余	B % A 结果为 0
++	自增	A++ 结果为 11
--	自减	A-- 结果为 9

实例

以下为算术运算的简单实例：

```
import Cocoa

var A = 10
var B = 20

print("A + B 结果为：\(A + B)")
print("A - B 结果为：\(A - B)")
print("A * B 结果为：\(A * B)")
print("B / A 结果为：\(B / A)")
A++
print("A++ 后 A 的值为 \(A)")
B--
print("B-- 后 B 的值为 \(B)")
```

以上程序执行结果为：

```
A + B 结果为：30
A - B 结果为：-10
A * B 结果为：200
B / A 结果为：2
A++ 后 A 的值为 11
B-- 后 B 的值为 19
```

比较运算符

以下表格列出了 Swift 语言支持的比较运算符，其中变量 A 为 10，变量 B 为 20：

运算符	描述	实例
==	等于	(A == B) 为 false。
!=	不等于	(A != B) 为 true。
>	大于	(A > B) 为 false。
<	小于	(A < B) 为 true。
>=	大于等于	(A >= B) 为 false。
<=	小于等于	(A <= B) 为 true。

实例

以下为比较运算的简单实例：

```
import Cocoa

var A = 10
var B = 20

print("A == B 结果为：\(A == B)")
print("A != B 结果为：\(A != B)")
print("A > B 结果为：\(A > B)")
print("A < B 结果为：\(A < B)")
print("A >= B 结果为：\(A >= B)")
print("A <= B 结果为：\(A <= B)")
```

以上程序执行结果为：

```
A == B 结果为：false
A != B 结果为：true
A > B 结果为：false
A < B 结果为：true
A >= B 结果为：false
A <= B 结果为：true
```

逻辑运算符

以下表格列出了 Swift 语言支持的逻辑运算符，其中变量 A 为 true，变量 B 为 false：

运算符	描述	实例
&&	逻辑与。如果运算符两侧都为 TRUE 则为 TRUE。	(A && B) 为 false。
||	逻辑或。 如果运算符两侧至少有一个为 TRUE 则为 TRUE。	(A || B) 为 true。
!	逻辑非。布尔值取反，使得true变false，false变true。	!(A && B) 为 true。

以下为逻辑运算的简单实例：

```
import Cocoa

var A = true
var B = false

print("A && B 结果为：\(A && B)")
print("A || B 结果为：\(A || B)")
print("!A 结果为：\(!A)")
print("!B 结果为：\(!B)")
```

以上程序执行结果为：

```
A && B 结果为：false
A || B 结果为：true
!A 结果为：false
!B 结果为：true
```

位运算符

位运算符用来对二进制位进行操作，~,&,|,^分别为取反，按位与与，按位与或，按位与异或运算，如下表实例：

p	q	p & q	p | q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

如果指定 A = 60; 及 B = 13; 两个变量对应的二进制为：

```
A = 0011 1100

B = 0000 1101
```

进行位运算：

运算符	描述	图解	实例
	按位与。按位与运算符对两个数		

&	进行操作，然后返回一个新的数，这个数的每个位都需要两个输入数的同一位都为1时才为1。	Input 1: 1 1 1 1 1 1 0 0 Result: 0 0 1 1 1 1 0 0 Input 2: 0 0 1 1 1 1 1 1	(A & B) 结果为 12, 二进制为 0000 1100
|	按位或。按位或运算符|比较两个数，然后返回一个新的数，这个数的每一位设置1的条件是两个输入数的同一位都不为0(即任意一个为1，或都为1)。	Input 1: 1 0 1 1 0 0 1 0 Result: 1 1 1 1 1 1 1 0 Input 2: 0 1 0 1 1 1 1 0	(A | B) 结果为 61, 二进制为 0011 1101
^	按位异或. 按位异或运算符^比较两个数，然后返回一个数，这个数的每个位设为1的条件是两个输入数的同一位不同，如果相同就设为0。	Input 1: 0 0 0 1 0 1 0 0 Result: 0 0 0 1 0 0 0 1 Input 2: 0 0 0 0 0 1 0 1	(A ^ B) 结果为 49, 二进制为 0011 0001
~	按位取反运算符~对一个操作数的每一位都取反。	Input 1: 0 0 0 0 1 1 1 1 Result: 1 1 1 1 0 0 0 0	(~A) 结果为 -61, 二进制为 1100 0011 in 2's complement form.
<<	按位左移。左移操作符（<<）将操作数的所有位向左移动指定的位数。	下图展示了11111111 << 1（11111111 左移一位）的结果。蓝色数字表示被移动位，灰色表示被丢弃位，空位用橙色的0填充。 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0	A << 2 结果为 240, 二进制为 1111 0000
	按位右移。右移	下图展示了11111111 >> 1（11111111 右移一位）的结果。蓝色数字表示被移动位，灰色表示被丢弃位，空位用橙色的0填	A >> 2 结果

>>	操作符（<<）将操作数的所有位向又移动指定的位数。	充。	A >> 2 结果为 15, 二进制为 0000 1111

以下为位运算的简单实例：

```
import Cocoa

var A = 60  // 二进制为 0011 1100
var B = 13 // 二进制为 0000 1101

print("A&B 结果为：\(A&B)")
print("A|B 结果为：\(A|B)")
print("A^B 结果为：\(A^B)")
print("~A 结果为：\(~A)")
```

以上程序执行结果为：

```
A&B 结果为：12
A|B 结果为：61
A^B 结果为：49
~A 结果为：-61
```

赋值运算

下表列出了 Swift 语言的基本赋值运算：

<table>
<tr><th>运算符</th><th>描述</th><th>实例</th></tr>
<tr><td>=</td><td>简单的赋值运算，指定右边操作数赋值给左边的操作数。</td><td>C = A + B 将 A + B 的运算结果赋值给 C</td></tr>
<tr><td>+=</td><td>相加后再赋值，将左右两边的操作数相加后再赋值给左边的操作数。</td><td>C += A 相当于 C = C + A</td></tr>
<tr><td>-=</td><td>相减后再赋值，将左右两边的操作数相减后再赋值给左边的操作数。</td><td>C -= A 相当于 C = C - A</td></tr>
<tr><td>*=</td><td>相乘后再赋值，将左右两边的操作数相乘后再赋值给左边的操作数。</td><td>C = A 相当于 C = C A</td></tr>
<tr><td>/=</td><td>相除后再赋值，将左右两边的操作数相除后再赋值给左边的操作数。</td><td>C /= A 相当于 C = C / A</td></tr>
<tr><td>%=</td><td>求余后再赋值，将左右两边的操作数求余后再赋值给左边的操作数。</td><td>C %= A is equivalent to C = C % A</td></tr>
<tr><td><<=</td><td>按位左移后再赋值</td><td>C <<= 2 相当于 C = C << 2</td></tr>
<tr><td>>>=</td><td>按位右移后再赋值</td><td>C >>= 2 相当于 C = C >> 2</td></tr>
<tr><td>&=</td><td>按位与运算后赋值</td><td>C &= 2 相当于 C = C & 2</td></tr>
<tr><td>^=</td><td>按位异或运算符后再赋值</td><td>C ^= 2 相当于 C = C ^ 2</td></tr>
<tr><td>|=</td><td>按位或运算后再赋值</td><td>C |= 2 相当于 C = C | 2</td></tr>
</table>

以下为赋值运算的简单实例：

```
import Cocoa

var A = 10
var B = 20
var C = 100

C = A + B
print("C 结果为：\(C)")

C += A
print("C 结果为：\(C)")

C -= A
print("C 结果为：\(C)")

C *= A
print("C 结果为：\(C)")

C /= A
print("C 结果为：\(C)")

//以下测试已注释，可去掉注释测试每个实例
/*
C %= A
print("C 结果为：\(C)")

C <<= A
print("C 结果为：\(C)")

C >>= A
print("C 结果为：\(C)")

C &= A
print("C 结果为：\(C)")

C ^= A
print("C 结果为：\(C)")

C |= A
print("C 结果为：\(C)")
*/
```

以上程序执行结果为：

```
C 结果为：30
C 结果为：40
C 结果为：30
C 结果为：300
C 结果为：30
```

区间运算符

Swift 提供了两个区间的运算符。

运算符	描述	实例
闭区间运算符	闭区间运算符（a...b）定义一个包含从a到b(包括a和b)的所有值的区间，b必须大于等于a。 闭区间运算符在迭代一个区间的所有值时是非常有用的，如在for-in循环中：	1...5 区间值为 1, 2, 3, 4 和 5
半开区间运算符	半开区间（a..<b）定义一个从a到b但不包括b的区间。 之所以称为半开区间，是因为该区间包含第一个值而不包括最后的值。<="" td=""></b）定义一个从a到b但不包括b的区间。>	1..< 5 区间值为 1, 2, 3, 和 4

以下为区间运算的简单实例：

```
import Cocoa

print("闭区间运算符:")
for index in 1...5 {
    print("\(index) * 5 = \(index * 5)")
}

print("半开区间运算符:")
for index in 1..<5 {
    print("\(index) * 5 = \(index * 5)")
}
```

以上程序执行结果为：

```
闭区间运算符:
1 * 5 = 5
2 * 5 = 10
3 * 5 = 15
4 * 5 = 20
5 * 5 = 25
半开区间运算符:
1 * 5 = 5
2 * 5 = 10
3 * 5 = 15
4 * 5 = 20
```

其他运算符

Swift 提供了其他类型的的运算符，如一元、二元和三元运算符。

- 一元运算符对单一操作对象操作（如 `-a` ）。一元运算符分前置运算符和后置运算符，前置运算符需紧跟在操作对象之前（如 `!b` ），后置运算符需紧跟在操作对象之后（如 `i++` ）。
- 二元运算符操作两个操作对象（如 `2 + 3` ），是中置的，因为它们出现在两个操作对象之间。
- 三元运算符操作三个操作对象，和 C 语言一样，Swift 只有一个三元运算符，就是三目运算符（ `a ? b : c` ）。

运算符	描述	实例
一元减	数字前添加 - 号前缀	-3 或 -4
一元加	数字钱添加 + 号前缀	+6 结果为 6
三元运算符	条件 ? X : Y	如果添加为 true，值为 X，否则为 Y

以下为一元、二元、三元的运算的简单实例：

```
import Cocoa

var A = 1
var B = 2
var C = true
var D = false
print("-A 的值为：\(-A)")
print("A + B 的值为：\(A + B)")
print("三元运算：\(C ? A : B )")
print("三元运算：\(D ? A : B )")
```

以上程序执行结果为：

```
-A 的值为：-1
A + B 的值为：3
三元运算：1
三元运算：2
```

运算符优先级

在一个表达式中可能包含多个有不同运算符连接起来的、具有不同数据类型的数据对象；由于表达式有多种运算，不同的运算顺序可能得出不同结果甚至出现错误运算错误，因为当表达式中含多种运算时，必须按一定顺序进行结合，才能保证运算的合理性和结果的正确性、唯一性。

优先级从上到下依次递减，最上面具有最高的优先级，逗号操作符具有最低的优先级。

相同优先级中，按结合顺序计算。大多数运算是从左至右计算，只有三个优先级是从右至左结合的，它们是单目运算符、条件运算符、赋值运算符。

基本的优先级需要记住：

- 指针最优，单目运算优于双目运算。如正负号。
- 先乘除（模），后加减。
- 先算术运算，后移位运算，最后位运算。请特别注意：1 << 3 + 2 & 7 等价于 (1 << (3 + 2))&7
- 逻辑运算最后计算

运算符类型	运算符	结合方向
表达式运算	() [] . expr++ expr--	左到右
一元运算符	& + - ! ~ ++expr --expr * / % + - >> << < > <= >= == !=	右到左
位运算符	& ^ | && ||	左到右
三元运算符	?:	右到左
赋值运算符	= += -= *= /= %= >>= <<= &= ^= |=	右到左
逗号	,	左到右

以下为运算符优先级简单实例：

```
import Cocoa

var A = 0

A = 2 + 3 * 4 % 5
print("A 的值为：\(A)")
```

以上程序执行结果为：

```
A 的值为：4
```

实例解析：

根据运算符优先级，可以将以上程序的运算解析为以下步骤，表达式相当于：

```
2 + ((3 * 4) % 5)
```

第一步计算：(3 * 4) = 12，所以表达式相当于：

```
2 + (12 % 5)
```

第二步计算 12 % 5 = 2，所以表达式相当于：

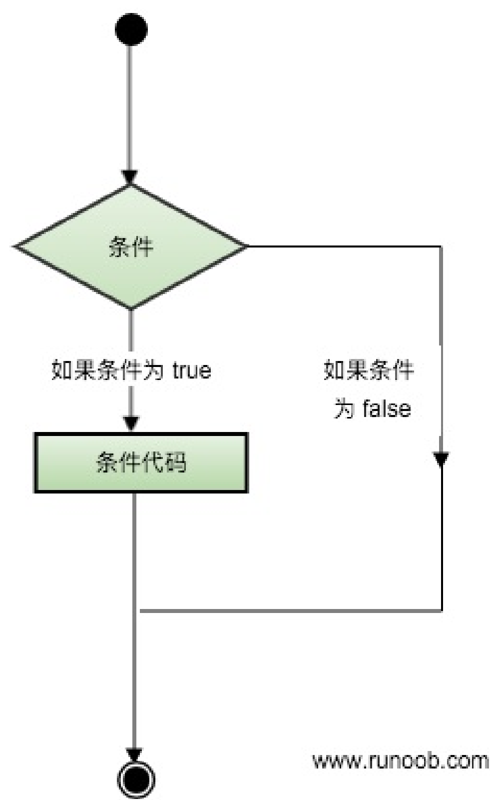
```
2 + 2
```

此时可以容易地看出计算的结果为 4。

Swift 条件语句

条件语句通过设定的一个或多个条件来执行程序，在条件为真时执行指定的语句，在条件为 false 时执行另外指定的语句。

可以通过下图来简单了解条件语句的执行过程:



Swift 提供了以下几种类型的条件语句：

语句	描述
if 语句	**if** 语句 由一个布尔表达式和一个或多个执行语句组成。
if...else 语句	**if** 语句 后可以有可选的 **else** 语句, **else** 语句在布尔表达式为 false 时执行。
if...else if...else 语句	**if** 后可以有可选的 **else if...else** 语句, **else if...else** 语句常用于多个条件判断。
内嵌 if 语句	你可以在 **if** 或 **else if** 中内嵌 **if** 或 **else if** 语句。
switch 语句	switch 语句允许测试一个变量等于多个值时的情况。

? : 运算符

我们已经在前面的章节中讲解了 条件运算符 **? :**，可以用来替代 **if...else** 语句。它的一般形式如下：

```
Exp1 ? Exp2 : Exp3;
```

其中，Exp1、Exp2 和 Exp3 是表达式。请注意，冒号的使用和位置。

? 表达式的值是由 Exp1 决定的。如果 Exp1 为真，则计算 Exp2 的值，结果即为整个 ? 表达式的值。如果 Exp1 为假，则计算 Exp3 的值，结果即为整个 ? 表达式的值。

Swift if 语句

一个 **if** 语句 由一个布尔表达式后跟一个或多个语句组成。

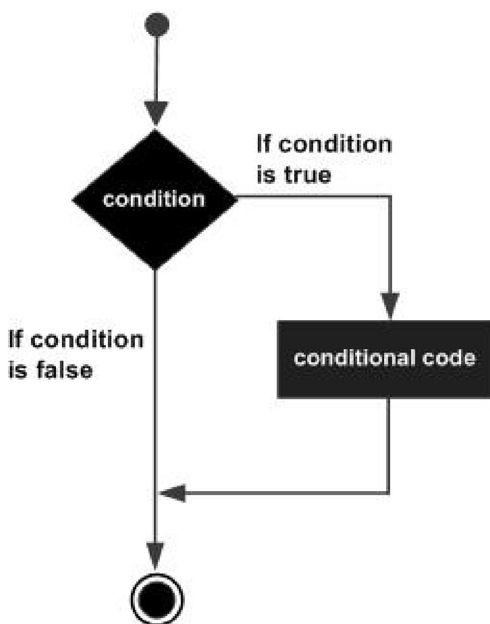
语法

Swift 语言中 **if** 语句的语法：

```
if boolean_expression {
   /* 如果布尔表达式为真将执行的语句 */
}
```

如果布尔表达式为 **true**，则 if 语句内的代码块将被执行。如果布尔表达式为 **false**，则 if 语句结束后的第一组代码（闭括号后）将被执行。

流程图



实例

```
import Cocoa

var varA:Int = 10;

/* 检测条件 */
if varA < 20 {
    /* 如果条件语句为 true 执行以下程序 */
    print("varA 小于 20");
}
print("varA 变量的值为 \(varA)");
```

当上面的代码被编译和执行时，它会产生下列结果：

```
varA 小于  20
varA 变量的值为  10
```

Swift if...else 语句

一个 **if** 语句 后可跟一个可选的 **else** 语句，else 语句在布尔表达式为 false 时执行。

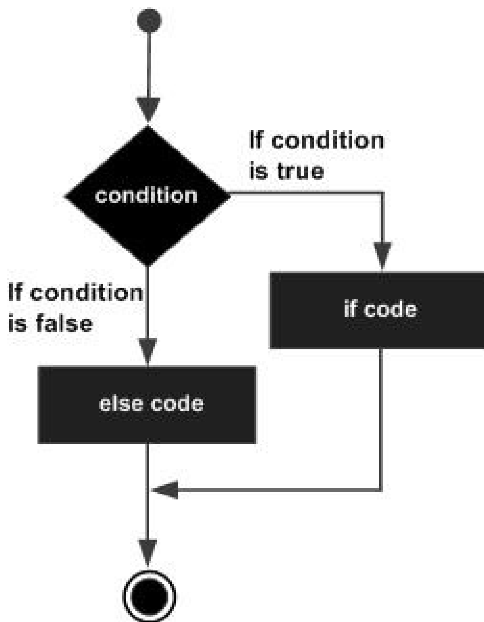
语法

Swift 语言中 **if...else** 语句的语法：

```
if boolean_expression {
   /* 如果布尔表达式为真将执行的语句 */
} else {
   /* 如果布尔表达式为假将执行的语句 */
}
```

如果布尔表达式为 **true**，则执行 **if** 块内的代码。如果布尔表达式为 **false**，则执行 **else** 块内的代码。

流程图



实例

```
import Cocoa

var varA:Int = 100;

/* 检测布尔条件 */
if varA < 20 {
   /* 如果条件为 true 执行以下语句 */
   print("varA 小于 20");
} else {
   /* 如果条件为 false 执行以下语句 */
   print("varA 大于 20");
}
print("varA 变量的值为 \(varA)");
```

当上面的代码被编译执行时，它会产生下列结果：

```
varA 大于 20
varA 变量的值为 100
```

Swift if...else if...else 语句

一个 **if** 语句 后可跟一个可选的 **else if...else** 语句，**else if...else** 语句 在测试多个条件语句时是非常有用的。

当你使用 if , else if , else 语句时需要注意以下几点：

- if 语句后可以有 0 个或 1 个 else，但是如果 有 else if 语句，else 语句需要在 else if 语句之后。
- if 语句后可以有 0 个或多个 else if 语句，else if 语句必须在 else 语句出现之前。
- 一旦 else 语句执行成功，其他的 else if 或 else 语句都不会执行。

语法

```
if boolean_expression_1 {
   /* 如果 boolean_expression_1 表达式为 true 则执行该语句 */
} else if boolean_expression_2 {
   /* 如果 boolean_expression_2 表达式为 true 则执行该语句 */
} else if boolean_expression_3 {
   /* 如果 boolean_expression_3 表达式为 true 则执行该语句 */
} else {
   /* 如果以上所有条件表达式都不为 true 则执行该语句 */
}
```

实例

```
import Cocoa

var varA:Int = 100;

/* 检测布尔条件 */
if varA == 20 {
    /* 如果条件为 true 执行以下语句 */
    print("varA 的值为 20");
} else if varA == 50 {
    /* 如果条件为 true 执行以下语句 */
    print("varA 的值为 50");
} else {
    /* 如果以上条件都为 false 执行以下语句 */
    print("没有匹配条件");
}
print("varA 变量的值为 \(varA)");
```

当上面的代码被编译执行时，它会产生下列结果：

```
没有匹配条件
varA 变量的值为  100
```

Swift 嵌套 if 语句

在 Swift 语言中，你可以在一个 if 或 else if 语句内使用另一个 if 或 else if 语句。

语法

Swift 语言中 嵌套 if 语句的语法：

```
import Cocoa

var varA:Int = 100;

/* 检测布尔条件 */
if varA == 20 {
    /* 如果条件为 true 执行以下语句 */
    print("varA 的值为 20");
} else if varA == 50 {
    /* 如果条件为 true 执行以下语句 */
    print("varA 的值为 50");
} else {
    /* 如果以上条件都为 false 执行以下语句 */
    print("没有匹配条件");
}
print("varA 变量的值为 \(varA)");
```

您可以嵌套 **else if...else**，方式与嵌套 *if* 语句相似。

实例

```
import Cocoa

var varA:Int = 100;
var varB:Int = 200;

/* 检测布尔条件 */
if varA == 100 {
   /* 如果条件为 true 执行以下语句 */
   print("第一个条件为 true");

   if varB == 200 {
      /* 如果条件为 true 执行以下语句 */
      print("第二个条件也是 true");
   }
}
print("varA 变量的值为 \(varA)");
print("varB 变量的值为 \(varB)");
```

当上面的代码被编译执行时，它会产生下列结果：

```
第一个条件为 true
第二个条件也是 true
varA 变量的值为 100
varB 变量的值为 200
```

Swift switch 语句

switch 语句允许测试一个变量等于多个值时的情况。 Swift 语言中只要匹配到 case 语句，则整个 switch 语句执行完成。

语法

Swift 语言中 switch 语句的语法：

```
switch expression {
   case expression1  :
      statement(s)
      fallthrough /* 可选 */
   case expression2, expression3  :
      statement(s)
      fallthrough /* 可选 */

   default : /* 可选 */
      statement(s);
}
```

一般在 switch 语句中不使用 fallthrough 语句。

这里我们需要注意 case 语句中如果没有使用 **fallthrough** 语句，则在执行当前的 case 语句后，switch 会终止，控制流将跳转到 switch 语句后的下一行。

如果使用了**fallthrough** 语句，则会继续执行之后的 case 或 default 语句，不论条件是否满足都会执行。

> 注意：在大多数语言中，switch 语句块中，case 要紧跟 break，否则 case 之后的语句会顺序运行，而在 Swift 语言中，默认是不会执行下去的，switch 也会终止。如果你想在 Swift 中让 case 之后的语句会按顺序继续运行，则需要使用 fallthrough 语句。

实例1

以下实例没有使用 fallthrough 语句：

```
import Cocoa

var index = 10

switch index {
   case 100  :
      print( "index 的值为 100")
   case 10,15  :
      print( "index 的值为 10 或 15")
   case 5  :
      print( "index 的值为 5")
   default :
      print( "默认 case")
}
```

当上面的代码被编译执行时，它会产生下列结果：

```
index 的值为  10  或  15
```

实例2

以下实例使用 fallthrough 语句：

```
import Cocoa

var index = 10

switch index {
   case 100  :
      print( "index 的值为 100")
      fallthrough
   case 10,15  :
      print( "index 的值为 10 或 15")
      fallthrough
   case 5  :
      print( "index 的值为 5")
   default :
      print( "默认 case")
}
```

当上面的代码被编译执行时，它会产生下列结果：

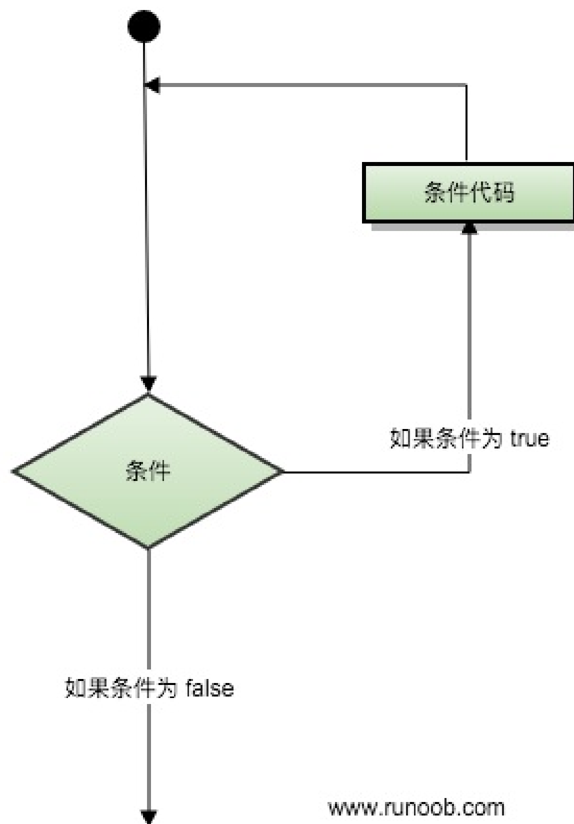
```
index 的值为  10  或  15
index 的值为  5
```

Swift 循环

有的时候，我们可能需要多次执行同一块代码。一般情况下，语句是按顺序执行的：函数中的第一个语句先执行，接着是第二个语句，依此类推。

编程语言提供了更为复杂执行路径的多种控制结构。

循环语句允许我们多次执行一个语句或语句组，下面是大多数编程语言中循环语句的流程图：



循环类型

Swift 语言提供了以下几种循环类型。点击链接查看每个类型的详细描述：

循环类型	描述
for-in	遍历一个集合里面的所有元素，例如由数字表示的区间、数组中的元素、字符串中的字符。
for 循环	用来重复执行一系列语句直到达成特定条件达成，一般通过在每次循环完成后增加计数器的值来实现。
while 循环	运行一系列语句，如果条件为true，会重复运行，直到条件变为false。
repeat...while 循环	类似 while 语句区别在于判断循环条件之前，先执行一次循环的代码块。

循环控制语句

循环控制语句改变你代码的执行顺序，通过它你可以实现代码的跳转。Swift 以下几种循环控制语句：

控制语句	描述
continue 语句	告诉一个循环体立刻停止本次循环迭代，重新开始下次循环迭代。
break 语句	中断当前循环。
fallthrough 语句	如果在一个case执行完后，继续执行下面的case，需要使用fallthrough(贯穿)关键字。

Swift for-in 循环

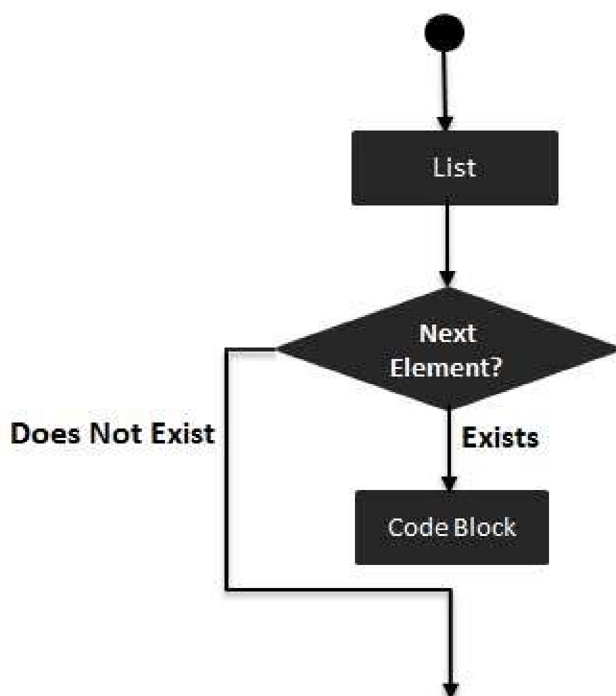
Swift for-in 循环用于遍历一个集合里面的所有元素，例如由数字表示的区间、数组中的元素、字符串中的字符。

语法

Swift for-in 循环的语法格式如下：

```
for index in var {
   循环体
}
```

流程图：



实例

```
import Cocoa

var someInts:[Int] = [10, 20, 30]

for index in someInts {
   print( "index 的值为 \(index)")
}
```

以上程序执行输出结果为：

```
index 的值为  10
index 的值为  20
index 的值为  30
```

Swift for 循环

Swift for 循环用来重复执行一系列语句直到达成特定条件，一般通过在每次循环完成后增加计数器的值来实现。

语法

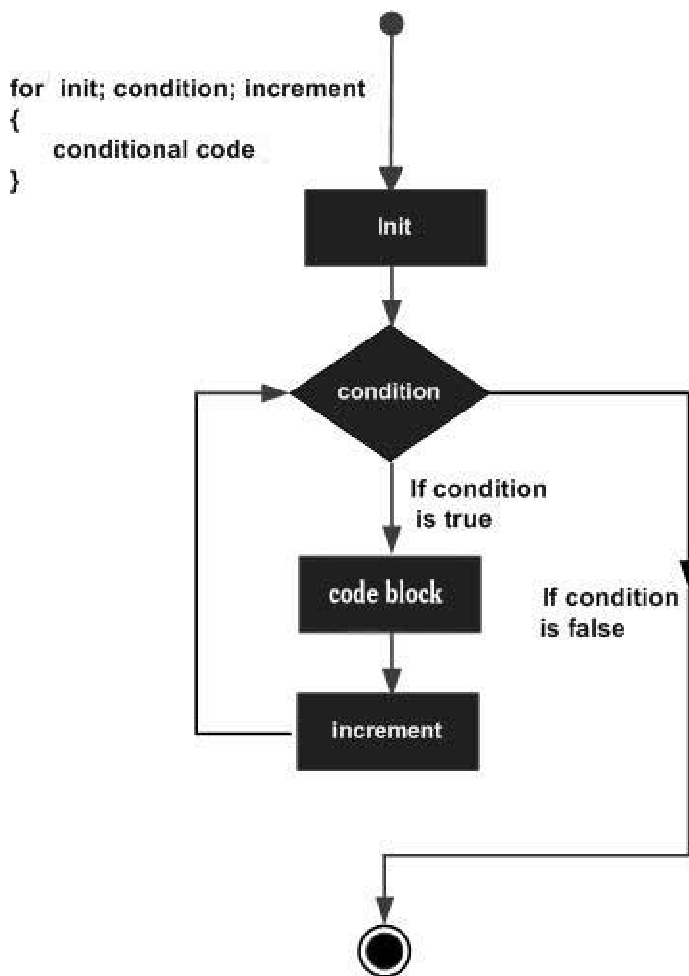
Swift for 循环的语法格式如下：

```
for init; condition; increment{
   循环体
}
```

参数解析：

1. **init** 会首先被执行，且只会执行一次。这一步允许您声明并初始化任何循环控制变量。您也可以不在这里写任何语句，只要有一个分号出现即可。
2. 接下来，会判断 **condition**。如果为真，则执行循环主体。如果为假，则不执行循环主体，且控制流会跳转到紧接着 for 循环的下一条语句。
3. 在执行完 for 循环主体后，控制流会跳回上面的 **increment** 语句。该语句允许您更新循环控制变量。该语句可以留空，只要在条件后有一个分号出现即可。
4. 条件再次被判断。如果为真，则执行循环，这个过程会不断重复（循环主体，然后增加步值，再然后重新判断条件）。在条件变为假时，for 循环终止。

流程图：



实例

```
import Cocoa

var someInts:[Int] = [10, 20, 30]

for var index = 0; index < 3; ++index {
   print( "索引 [\(index)] 对应的值为 \(someInts[index])")
}
```

以上程序执行输出结果为：

```
索引 [0] 对应的值为 10
索引 [1] 对应的值为 20
索引 [2] 对应的值为 30
```

Swift While 循环

Swift while循环从计算单一条件开始。如果条件为true，会重复运行一系列语句，直到条件变为false。

语法

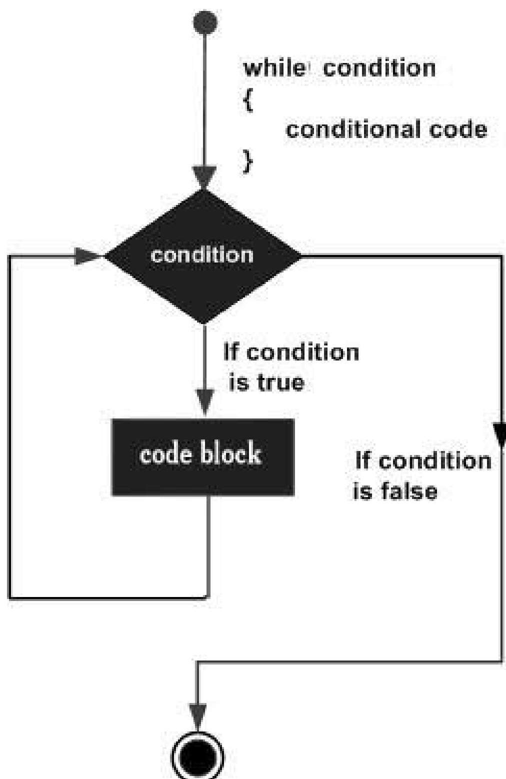
Swift while 循环的语法格式如下：

```
while condition
{
   statement(s)
}
```

语法中的 **statement(s)** 可以是一个语句或者一个语句块。 **condition** 可以是一个表达式。如果条件为true，会重复运行一系列语句，直到条件变为false。

数字 0, 字符串 '0' 和 "", 空的 list(), 及未定义的变量都为 **false** ， 其他的则都为 **true**。true 取反使用 **!** 号或 **not**，取反后返回 false。

流程图：



实例

```
import Cocoa

var index = 10

while index < 20
{
   print( "index 的值为 \(index)")
   index = index + 1
}
```

以上程序执行输出结果为：

```
index 的值为 10
index 的值为 11
index 的值为 12
index 的值为 13
index 的值为 14
index 的值为 15
index 的值为 16
index 的值为 17
index 的值为 18
index 的值为 19
```

Swift repeat...while 循环

Swift repeat...while 循环不像 for 和 while 循环在循环体开始执行前先判断条件语句，而是在循环执行结束时判断条件是否符合。

语法

Swift repeat...while 循环的语法格式如下：

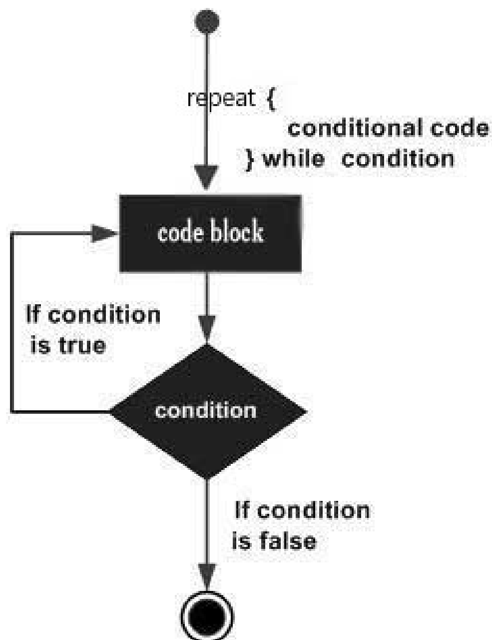
```
repeat
{
   statement(s);
}while( condition );
```

请注意，条件表达式出现在循环的尾部，所以循环中的 statement(s) 会在条件被测试之前至少执行一次。

如果条件为 true，控制流会跳转回上面的 repeat，然后重新执行循环中的 statement(s)。这个过程会不断重复，直到给定条件变为 false 为止。

数字 0, 字符串 '0' 和 "", 空的 list(), 及未定义的变量都为 **false**，其他的则都为 **true**。true 取反使用 **!** 号或 **not**，取反后返回 false。

流程图：



实例

```
import Cocoa

var index = 15

repeat{
    print( "index 的值为 \(index)")
    index = index + 1
}while index < 20
```

以上程序执行输出结果为：

```
index 的值为  15
index 的值为  16
index 的值为  17
index 的值为  18
index 的值为  19
```

Swift Continue 语句

Swift continue语句告诉一个循环体立刻停止本次循环迭代，重新开始下次循环迭代。

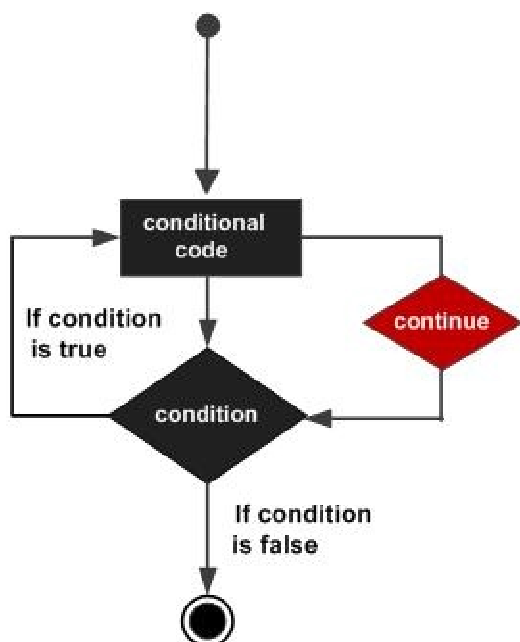
对于 **for** 循环，**continue** 语句执行后自增语句仍然会执行。对于 **while** 和 **do...while** 循环，**continue** 语句重新执行条件判断语句。

语法

Swift continue语句的语法格式如下：

```
continue
```

流程图：



实例

```
import Cocoa

var index = 10

repeat{
   index = index + 1

   if( index == 15 ){ // index 等于 15 时跳过
      continue
   }
   print( "index 的值为 \(index)")
}while index < 20
```

以上程序执行输出结果为：

```
index 的值为 11
index 的值为 12
index 的值为 13
index 的值为 14
index 的值为 16
index 的值为 17
index 的值为 18
index 的值为 19
index 的值为 20
```

Swift Break 语句

Swift break语句会立刻结束整个控制流的执行。

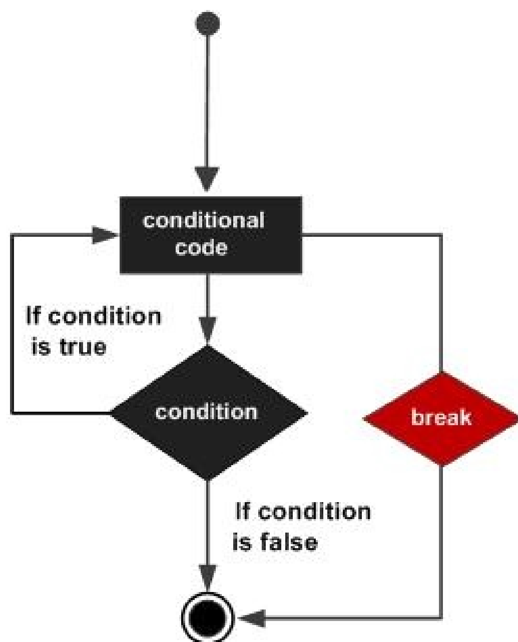
如果您使用的是嵌套循环（即一个循环内嵌套另一个循环），break 语句会停止执行最内层的循环，然后开始执行该块之后的下一行代码。

语法

Swift break语句的语法格式如下：

```
break
```

流程图：



实例

```
import Cocoa

var index = 10

repeat{
    index = index + 1

    if( index == 15 ){  // index 等于 15 时终止循环
        break
    }
    print( "index 的值为 \(index)")
}while index < 20
```

以上程序执行输出结果为：

```
index 的值为 11
index 的值为 12
index 的值为 13
index 的值为 14
```

Swift Fallthrough 语句

Swift fallthrough 语句让 case 之后的语句会按顺序继续运行，且不论条件是否满足都会执行。

Swift 中的 switch 不会从上一个 case 分支落入到下一个 case 分支中。只要第一个匹配到的 case 分支完成了它需要执行的语句，整个switch代码块完成了它的执行。

> 注意：在大多数语言中，switch 语句块中，case 要紧跟 break，否则 case 之后的语句会顺序运行，而在 Swift 语言中，默认是不会执行下去的，switch 也会终止。如果你想在 Swift 中让 case 之后的语句会按顺序继续运行，则需要使用 fallthrough 语句。

语法

Swift fallthrough 语句的语法格式如下：

```
fallthrough
```

一般在 switch 语句中不使用 fallthrough 语句。

实例1

以下实例没有使用 fallthrough 语句：

```
import Cocoa

var index = 10

switch index {
   case 100  :
      print( "index 的值为 100")
   case 10,15  :
      print( "index 的值为 10 或 15")
   case 5  :
      print( "index 的值为 5")
   default :
      print( "默认 case")
}
```

当上面的代码被编译执行时，它会产生下列结果：

```
index 的值为  10  或  15
```

实例2

以下实例使用 fallthrough 语句：

```
import Cocoa

var index = 10

switch index {
   case 100  :
      print( "index 的值为 100")
      fallthrough
   case 10,15  :
      print( "index 的值为 10 或 15")
      fallthrough
   case 5  :
      print( "index 的值为 5")
   default :
      print( "默认 case")
}
```

当上面的代码被编译执行时，它会产生下列结果：

```
index 的值为  10  或  15
index 的值为  5
```

Swift 字符串

Swift 字符串是一系列字符的集合。例如 "Hello, World!" 这样的有序的字符类型的值的集合，它的数据类型为 **String**。

创建字符串

你可以通过使用字符串字面量或 String 类的实例来创建一个字符串：

```
import Cocoa

// 使用字符串字面量
var stringA = "Hello, World!"
print( stringA )

// String 实例化
var stringB = String("Hello, World!")
print( stringB )
```

以上程序执行输出结果为：

```
Hello, World!
Hello, World!
```

空字符串

你可以使用空的字符串字面量赋值给变量或初始化一个String类的实例来初始值一个空的字符串。 我们可以使用字符串属性 isEmpty 来判断字符串是否为空：

```
import Cocoa

// 使用字符串字面量创建空字符串
var stringA = ""

if stringA.isEmpty {
   print( "stringA 是空的" )
} else {
   print( "stringA 不是空的" )
}

// 实例化 String 类来创建空字符串
let stringB = String()

if stringB.isEmpty {
   print( "stringB 是空的" )
} else {
   print( "stringB 不是空的" )
}
```

以上程序执行输出结果为：

```
stringA 是空的
stringB 是空的
```

字符串常量

你可以将一个字符串赋值给一个变量或常量，变量是可修改的，常量是不可修改的。

```
import Cocoa

// stringA 可被修改
var stringA = "菜鸟教程："
stringA += "http://www.runoob.com"
print( stringA )

// stringB 不能修改
let stringB = String("菜鸟教程：")
stringB += "http://www.runoob.com"
print( stringB )
```

以上程序执行输出结果会报错，以为 stringB 为常量是不能被修改的：

```
error: left side of mutating operator isn't mutable: 'stringB' is a
stringB += "http://www.runoob.com"
```

字符串中插入值

字符串插值是一种构建新字符串的方式，可以在其中包含常量、变量、字面量和表达式。 您插入的字符串字面量的每一项都在以反斜线为前缀的圆括号中：

```
import Cocoa

var varA   = 20
let constA = 100
var varC:Float = 20.0

var stringA = "\(varA) 乘于 \(constA) 等于 \(varC * 100)"
print( stringA )
```

以上程序执行输出结果为：

```
20 乘于 100 等于 2000.0
```

字符串连接

字符串可以通过 **+** 号来连接，实例如下：

```
import Cocoa

let constA = "菜鸟教程："
let constB = "http://www.runoob.com"

var stringA = constA + constB

print( stringA )
```

以上程序执行输出结果为：

```
菜鸟教程：http://www.runoob.com
```

字符串长度

字符串长度使用 **String.characters.count** 属性来计算，实例如下：

```
import Cocoa

var varA   = "www.runoob.com"

print( "\(varA), 长度为 \(varA.characters.count)" )
```

以上程序执行输出结果为：

```
www.runoob.com, 长度为 14
```

字符串比较

你可以使用 **==** 来比较两个字符串是否相等：

```
import Cocoa

var varA   = "Hello, Swift!"
var varB   = "Hello, World!"

if varA == varB {
   print( "\(varA) 与 \(varB) 是相等的" )
} else {
   print( "\(varA) 与 \(varB) 是不相等的" )
}
```

以上程序执行输出结果为：

```
Hello, Swift! 与 Hello, World! 是不相等的
```

Unicode 字符串

Unicode 是一个国际标准，用于文本的编码，Swift 的 String 类型是基于 Unicode 建立的。你可以循环迭代出字符串中 UTF-8 与 UTF-16 的编码，实例如下：

```
import Cocoa

var unicodeString   = "菜鸟教程"

print("UTF-8 编码: ")
for code in unicodeString.utf8 {
   print("\(code) ")
}

print("\n")

print("UTF-16 编码: ")
for code in unicodeString.utf16 {
   print("\(code) ")
}
```

以上程序执行输出结果为：

```
UTF-8 编码:
232
143
156
233
184
159
230
149
153
231
168
139
UTF-16 编码:
33756
40479
25945
31243
```

字符串函数及运算符

Swift 支持以下几种字符串函数及运算符：

函数/运算符	描述
isEmpty	判断字符串是否为空，返回布尔值
hasPrefix(prefix: String)	检查字符串是否拥有特定后缀
hasSuffix(suffix: String)	检查字符串是否拥有特定后缀。

Int(String)	转换字符串数字为整型。 实例:
	`let myString: String = "256"`
	`let myInt: Int? = Int(myString)`

String.characters.count	计算字符串的长度
utf8	您可以通过遍历 String 的 utf8 属性来访问它的 UTF-8 编码
utf16	您可以通过遍历 String 的 utf8 属性来访问它的 UTF-16 编码
unicodeScalars	您可以通过遍历String值的unicodeScalars属性来访问它的 Unicode 标量编码.
+	连接两个字符串，并返回一个新的字符串
+=	连接操作符两边的字符串并将新字符串赋值给左边的操作符变量
==	判断两个字符串是否相等
<	比较两个字符串，对两个字符串的字母逐一比较。
!=	比较两个字符串是否不相等。

Swift 字符(Character)

Swift 的字符是一个单一的字符字符串字面量，数据类型为 Character。

以下实例列出了两个字符实例：

```
import Cocoa

let char1: Character = "A"
let char2: Character = "B"

print("char1 的值为 \(char1)")
print("char2 的值为 \(char2)")
```

以上程序执行输出结果为：

```
char1 的值为 A
char2 的值为 B
```

如果你想在 Character（字符） 类型的常量中存储更多的字符，则程序执行会报错，如下所示：

```
import Cocoa

// Swift 中以下赋值会报错
let char: Character = "AB"

print("Value of char \(char)")
```

以上程序执行输出结果为：

```
error: cannot convert value of type 'String' to specified type 'Cha
let char: Character = "AB"
```

空字符变量

Swift 中不能创建空的 Character（字符） 类型变量或常量：

```
import Cocoa

// Swift 中以下赋值会报错
let char1: Character = ""
var char2: Character = ""

print("char1 的值为 \(char1)")
print("char2 的值为 \(char2)")
```

以上程序执行输出结果为：

```
 error: cannot convert value of type 'String' to specified type 'Ch
let char1: Character = ""
                       ^~
error: cannot convert value of type 'String' to specified type 'Cha
var char2: Character = ""
```

遍历字符串中的字符

Swift 的 String 类型表示特定序列的 Character（字符） 类型值的集合。 每一个字符值代表一个 Unicode 字符。

您可通过for-in循环来遍历字符串中的characters属性来获取每一个字符的值：

```
import Cocoa

for ch in "Hello".characters {
   print(ch)
}
```

以上程序执行输出结果为：

```
H
e
l
l
o
```

字符串连接字符

以下实例演示了使用 String 的 append() 方法来实现字符串连接字符：

```
import Cocoa

var varA:String = "Hello "
let varB:Character = "G"

varA.append( varB )

print("varC  =  \(varA)")
```

以上程序执行输出结果为：

```
varC  =  Hello G
```

Swift 数组

Swift 数组使用有序列表存储同一类型的多个值。相同的值可以多次出现在一个数组的不同位置中。

Swift 数组会强制检测元素的类型，如果类型不同则会报错，Swift 数组应该遵循像Array<Element>这样的形式，其中Element是这个数组中唯一允许存在的数据类型。

如果创建一个数组，并赋值给一个变量，则创建的集合就是可以修改的。这意味着在创建数组后，可以通过添加、删除、修改的方式改变数组里的项目。如果将一个数组赋值给常量，数组就不可更改，并且数组的大小和内容都不可以修改。

创建数组

我们可以使用构造语法来创建一个由特定数据类型构成的空数组：

```
var someArray = [SomeType]()
```

以下是创建一个初始化大小数组的语法：

```
var someArray = [SomeType](count: NumbeOfElements, repeatedValue: I
```

以下实例创建了一个类型为 Int ，大小为 3，初始值为 0 的空数组：

```
var someInts = [Int](count: 3, repeatedValue: 0)
```

以下实例创建了含有三个元素的数组：

```
var someInts:[Int] = [10, 20, 30]
```

访问数组

我们可以根据数组的索引来访问数组的元素，语法如下：

```
var someVar = someArray[index]
```

index 索引从 0 开始，及索引 0 对应第一个元素，索引 1 对应第二个元素，以此类推。

我们可以通过以下实例来学习如何创建，初始化，访问数组：

```
import Cocoa

var someInts = [Int](count: 3, repeatedValue: 10)

var someVar = someInts[0]

print( "第一个元素的值 \(someVar)" )
print( "第二个元素的值 \(someInts[1])" )
print( "第三个元素的值 \(someInts[2])" )
```

以上程序执行输出结果为：

```
第一个元素的值 10
第二个元素的值 10
第三个元素的值 10
```

修改数组

你可以使用 append() 方法或者赋值运算符 += 在数组末尾添加元素，如下所示，我们初始化一个数组，并向其添加元素：

```
import Cocoa

var someInts = [Int]()

someInts.append(20)
someInts.append(30)
someInts += [40]

var someVar = someInts[0]

print( "第一个元素的值 \(someVar)" )
print( "第二个元素的值 \(someInts[1])" )
print( "第三个元素的值 \(someInts[2])" )
```

以上程序执行输出结果为：

```
第一个元素的值 20
第二个元素的值 30
第三个元素的值 40
```

我们也可以通过索引修改数组元素的值：

```
import Cocoa

var someInts = [Int]()

someInts.append(20)
someInts.append(30)
someInts += [40]

// 修改最后一个元素
someInts[2] = 50

var someVar = someInts[0]

print( "第一个元素的值 \(someVar)" )
print( "第二个元素的值 \(someInts[1])" )
print( "第三个元素的值 \(someInts[2])" )
```

以上程序执行输出结果为：

```
第一个元素的值 20
第二个元素的值 30
第三个元素的值 50
```

遍历数组

我们可以使用for-in循环来遍历所有数组中的数据项：

```
import Cocoa

var someStrs = [String]()

someStrs.append("Apple")
someStrs.append("Amazon")
someStrs.append("Runoob")
someStrs += ["Google"]

for item in someStrs {
   print(item)
}
```

以上程序执行输出结果为：

```
Apple
Amazon
Runoob
Google
```

如果我们同时需要每个数据项的值和索引值，可以使用 String 的 enumerate() 方法来进行数组遍历。实例如下：

```
import Cocoa

var someStrs = [String]()

someStrs.append("Apple")
someStrs.append("Amazon")
someStrs.append("Runoob")
someStrs += ["Google"]

for (index, item) in someStrs.enumerate() {
   print("在 index = \(index) 位置上的值为 \(item)")
}
```

以上程序执行输出结果为：

```
在 index = 0 位置上的值为 Apple
在 index = 1 位置上的值为 Amazon
在 index = 2 位置上的值为 Runoob
在 index = 3 位置上的值为 Google
```

合并数组

我们可以使用加法操作符（+）来合并两种已存在的相同类型数组。新数组的数据类型会从两个数组的数据类型中推断出来：

```
import Cocoa

var intsA = [Int](count:2, repeatedValue: 2)
var intsB = [Int](count:3, repeatedValue: 1)

var intsC = intsA + intsB

for item in intsC {
   print(item)
}
```

以上程序执行输出结果为：

```
2
2
1
1
1
```

count 属性

我们可以使用 count 属性来计算数组元素个数：

```
import Cocoa

var intsA = [Int](count:2, repeatedValue: 2)
var intsB = [Int](count:3, repeatedValue: 1)

var intsC = intsA + intsB

print("intsA 元素个数为 \(intsA.count)")
print("intsB 元素个数为 \(intsB.count)")
print("intsC 元素个数为 \(intsC.count)")
```

以上程序执行输出结果为：

```
intsA 元素个数为 2
intsB 元素个数为 3
intsC 元素个数为 5
```

isEmpty 属性

我们可以通过只读属性 isEmpty 来判断数组是否为空，返回布尔值：

```
import Cocoa

var intsA = [Int](count:2, repeatedValue: 2)
var intsB = [Int](count:3, repeatedValue: 1)
var intsC = [Int]()

print("intsA.isEmpty = \(intsA.isEmpty)")
print("intsB.isEmpty = \(intsB.isEmpty)")
print("intsC.isEmpty = \(intsC.isEmpty)")
```

以上程序执行输出结果为：

```
intsA.isEmpty = false
intsB.isEmpty = false
intsC.isEmpty = true
```

Swift 字典

Swift 字典用来存储无序的相同类型数据的集合，Swift 数组会强制检测元素的类型，如果类型不同则会报错。

Swift 字典每个值（value）都关联唯一的键（key），键作为字典中的这个值数据的标识符。

和数组中的数据项不同，字典中的数据项并没有具体顺序。我们在需要通过标识符（键）访问数据的时候使用字典，这种方法很大程度上和我们在现实世界中使用字典查字义的方法一样。

Swift 字典的key没有类型限制可以是整型或字符串，但必须是唯一的。

如果创建一个字典，并赋值给一个变量，则创建的字典就是可以修改的。这意味着在创建字典后，可以通过添加、删除、修改的方式改变字典里的项目。如果将一个字典赋值给常量，字典就不可修改，并且字典的大小和内容都不可以修改。

创建字典

我们可以使用以下语法来创建一个特定类型的空字典：

```
var someDict =  [KeyType: ValueType]()
```

以下是创建一个空字典，键的类型为 Int，值的类型为 String 的简单语法：

```
var someDict = [Int: String]()
```

以下为创建一个字典的实例：

```
var someDict:[Int:String] = [1:"One", 2:"Two", 3:"Three"]
```

访问字典

我们可以根据字典的索引来访问数组的元素，语法如下：

```
var someVar = someDict[key]
```

我们可以通过以下实例来学习如何创建，初始化，访问字典:

```
import Cocoa

var someDict:[Int:String] = [1:"One", 2:"Two", 3:"Three"]

var someVar = someDict[1]

print( "key = 1 的值为 \(someVar)" )
print( "key = 2 的值为 \(someDict[2])" )
print( "key = 3 的值为 \(someDict[3])" )
```

以上程序执行输出结果为：

```
key = 1 的值为 Optional("One")
key = 2 的值为 Optional("Two")
key = 3 的值为 Optional("Three")
```

修改字典

我们可以使用 **updateValue(forKey:)** 增加或更新字典的内容。如果 key 不存在，则添加值，如果存在则修改 key 对应的值。updateValue(_:forKey:)方法返回Optional值。实例如下：

```
import Cocoa

var someDict:[Int:String] = [1:"One", 2:"Two", 3:"Three"]

var oldVal = someDict.updateValue("One 新的值", forKey: 1)

var someVar = someDict[1]

print( "key = 1 旧的值 \(oldVal)" )
print( "key = 1 的值为 \(someVar)" )
print( "key = 2 的值为 \(someDict[2])" )
print( "key = 3 的值为 \(someDict[3])" )
```

以上程序执行输出结果为：

```
key = 1 旧的值 Optional("One")
key = 1 的值为 Optional("One 新的值")
key = 2 的值为 Optional("Two")
key = 3 的值为 Optional("Three")
```

你也可以通过指定的 key 来修改字典的值，如下所示：

```
import Cocoa

var someDict:[Int:String] = [1:"One", 2:"Two", 3:"Three"]

var oldVal = someDict[1]
someDict[1] = "One 新的值"
var someVar = someDict[1]

print( "key = 1 旧的值 \(oldVal)" )
print( "key = 1 的值为 \(someVar)" )
print( "key = 2 的值为 \(someDict[2])" )
print( "key = 3 的值为 \(someDict[3])" )
```

以上程序执行输出结果为：

```
key = 1 旧的值 Optional("One")
key = 1 的值为 Optional("One 新的值")
key = 2 的值为 Optional("Two")
key = 3 的值为 Optional("Three")
```

移除 Key-Value 对

我们可以使用 **removeValueForKey()** 方法来移除字典 key-value 对。如果 key 存在该方法返回移除的值，如果不存在返回 nil 。实例如下：

```
import Cocoa

var someDict:[Int:String] = [1:"One", 2:"Two", 3:"Three"]

var removedValue = someDict.removeValueForKey(2)

print( "key = 1 的值为 \(someDict[1])" )
print( "key = 2 的值为 \(someDict[2])" )
print( "key = 3 的值为 \(someDict[3])" )
```

以上程序执行输出结果为：

```
key = 1 的值为 Optional("One")
key = 2 的值为 nil
key = 3 的值为 Optional("Three")
```

你也可以通过指定键的值为 nil 来移除 key-value（键-值）对。实例如下:

```
import Cocoa

var someDict:[Int:String] = [1:"One", 2:"Two", 3:"Three"]

someDict[2] = nil

print( "key = 1 的值为 \(someDict[1])" )
print( "key = 2 的值为 \(someDict[2])" )
print( "key = 3 的值为 \(someDict[3])" )
```

以上程序执行输出结果为：

```
key = 1 的值为 Optional("One")
key = 2 的值为 nil
key = 3 的值为 Optional("Three")
```

遍历字典

我们可以使用 **for-in** 循环来遍历某个字典中的键值对。实例如下:

```
import Cocoa

var someDict:[Int:String] = [1:"One", 2:"Two", 3:"Three"]

for (key, value) in someDict {
   print("字典 key \(key) -  字典 value \(value)")
}
```

以上程序执行输出结果为：

```
字典 key 2 -  字典 value Two
字典 key 3 -  字典 value Three
字典 key 1 -  字典 value One
```

我们也可以使用enumerate()方法来进行字典遍历，返回的是字典的索引及 (key, value) 对，实例如下:

```
import Cocoa

var someDict:[Int:String] = [1:"One", 2:"Two", 3:"Three"]

for (key, value) in someDict.enumerate() {
    print("字典 key \(key) -  字典 (key, value) 对 \(value)")
}
```

以上程序执行输出结果为：

```
字典 key 0 -  字典 (key, value) 对 (2, "Two")
字典 key 1 -  字典 (key, value) 对 (3, "Three")
字典 key 2 -  字典 (key, value) 对 (1, "One")
```

字典转换为数组

你可以提取字典的键值(key-value)对，并转换为独立的数组。实例如下：

```
import Cocoa

var someDict:[Int:String] = [1:"One", 2:"Two", 3:"Three"]

let dictKeys = [Int](someDict.keys)
let dictValues = [String](someDict.values)

print("输出字典的键(key)")

for (key) in dictKeys {
    print("\(key)")
}

print("输出字典的值(value)")

for (value) in dictValues {
    print("\(value)")
}
```

以上程序执行输出结果为：

```
输出字典的键(key)
2
3
1
输出字典的值(value)
Two
Three
One
```

count 属性

我们可以使用只读的 **count** 属性来计算字典有多少个键值对：

```
import Cocoa

var someDict1:[Int:String] = [1:"One", 2:"Two", 3:"Three"]
var someDict2:[Int:String] = [4:"Four", 5:"Five"]

print("someDict1 含有 \(someDict1.count) 个键值对")
print("someDict2 含有 \(someDict2.count) 个键值对")
```

以上程序执行输出结果为：

```
someDict1 含有 3 个键值对
someDict2 含有 2 个键值对
```

isEmpty 属性

Y我们可以通过只读属性 **isEmpty** 来判断字典是否为空，返回布尔值:

```
import Cocoa

var someDict1:[Int:String] = [1:"One", 2:"Two", 3:"Three"]
var someDict2:[Int:String] = [4:"Four", 5:"Five"]
var someDict3:[Int:String] = [Int:String]()

print("someDict1 = \(someDict1.isEmpty)")
print("someDict2 = \(someDict2.isEmpty)")
print("someDict3 = \(someDict3.isEmpty)")
```

以上程序执行输出结果为：

```
someDict1 = false
someDict2 = false
someDict3 = true
```

Swift 函数

Swift 函数用来完成特定任务的独立的代码块。

Swift使用一个统一的语法来表示简单的C语言风格的函数到复杂的Objective-C语言风格的方法。

- 函数声明: 告诉编译器函数的名字，返回类型及参数。
- 函数定义: 提供了函数的实体。

Swift 函数包含了参数类型及返回值类型：

函数定义

Swift 定义函数使用关键字 **func**。

定义函数的时候，可以指定一个或多个输入参数和一个返回值类型。

每个函数都有一个函数名来描述它的功能。通过函数名以及对应类型的参数值来调用这个函数。函数的参数传递的顺序必须与参数列表相同。

函数的实参传递的顺序必须与形参列表相同，**->** 后定义函数的返回值类型。

语法

```
func funcname(形参) -> returntype
{
   Statement1
   Statement2
   ……
   Statement N
   return parameters
}
```

实例

以下我们定义了一个函数名为 runoob 的函数，形参的数据类型为 String，返回值也为 String：

```
import Cocoa

func runoob(site: String) -> String {
    return site
}
print(runoob("www.runoob.com"))
```

以上程序执行输出结果为：

```
www.runoob.com
```

函数调用

我们可以通过函数名以及对应类型的参数值来调用函数，函数的参数传递的顺序必须与参数列表相同。

以下我们定义了一个函数名为 runoob 的函数，形参 site 的数据类型为 String，之后我们调用函数传递的实参也必须 String 类型，实参传入函数体后，将直接返回，返回的数据类型为 String。

```
import Cocoa

func runoob(site: String) -> String {
    return site
}
print(runoob("www.runoob.com"))
```

以上程序执行输出结果为：

```
www.runoob.com
```

函数参数

函数可以接受一个或者多个参数，我们也可以使用元组（tuple）向函数传递一个或多个参数：

```
import Cocoa

func mult(no1: Int, no2: Int) -> Int {
   return no1*no2
}
print(mult(2, no2:20))
print(mult(3, no2:15))
print(mult(4, no2:30))
```

以上程序执行输出结果为：

```
40
45
120
```

不带参数函数

我们可以创建不带参数的函数。

语法：

```
func funcname() -> datatype {
   return datatype
}
```

实例

```
import Cocoa

func sitename() -> String {
    return "菜鸟教程"
}
print(sitename())
```

以上程序执行输出结果为：

```
菜鸟教程
```

元组作为函数返回值

函数返回值类型可以是字符串，整型，浮点型等。

元组与数组类似，不同的是，元组中的元素可以是任意类型，使用的是圆括号。

你可以用元组（tuple）类型让多个值作为一个复合值从函数中返回。

下面的这个例子中，定义了一个名为minMax(_:)的函数，作用是在一个Int数组中找出最小值与最大值。

```
import Cocoa

func minMax(array: [Int]) -> (min: Int, max: Int) {
    var currentMin = array[0]
    var currentMax = array[0]
    for value in array[1..<array.count] {
        if value < currentMin {
            currentMin = value
        } else if value > currentMax {
            currentMax = value
        }
    }
    return (currentMin, currentMax)
}

let bounds = minMax([8, -6, 2, 109, 3, 71])
print("最小值为 \(bounds.min) ，最大值为 \(bounds.max)")
```

minMax(_:)函数返回一个包含两个Int值的元组，这些值被标记为min和max，以便查询函数的返回值时可以通过名字访问它们。

以上程序执行输出结果为：

```
最小值为 -6 ，最大值为 109
```

如果你不确定返回的元组一定不为nil，那么你可以返回一个可选的元组类型。

你可以通过在元组类型的右括号后放置一个问号来定义一个可选元组，例如(Int, Int)?或(String, Int, Bool)?

> 注意 可选元组类型如 `(Int, Int)?` 与元组包含可选类型如 `(Int?, Int?)` 是不同的.可选的元组类型，整个元组是可选的，而不只是元组中的每个元素值。

前面的 `minMax(_:)` 函数返回了一个包含两个 `Int` 值的元组。但是函数不会对传入的数组执行任何安全检查，如果 `array` 参数是一个空数组，如上定义的 `minMax(_:)` 在试图访问 `array[0]` 时会触发一个运行时错误。

为了安全地处理这个"空数组"问题，将 `minMax(_:)` 函数改写为使用可选元组返回类型，并且当数组为空时返回 `nil` ：

```
import Cocoa

func minMax(array: [Int]) -> (min: Int, max: Int)? {
    if array.isEmpty { return nil }
    var currentMin = array[0]
    var currentMax = array[0]
    for value in array[1..<array.count] {
        if value < currentMin {
            currentMin = value
        } else if value > currentMax {
            currentMax = value
        }
    }
    return (currentMin, currentMax)
}
if let bounds = minMax([8, -6, 2, 109, 3, 71]) {
    print("最小值为 \(bounds.min)，组大值为 \(bounds.max)")
}
```

以上程序执行输出结果为：

```
最小值为 -6，组大值为 109
```

没有返回值函数

下面是 runoob(_:) 函数的另一个版本，这个函数接收菜鸟教程官网网址参数，没有指定返回值类型，并直接输出 String 值，而不是返回它：

```
import Cocoa

func runoob(site: String) {
    print("菜鸟教程官网：\(site)")
}
runoob("http://www.runoob.com")
```

以上程序执行输出结果为：

```
菜鸟教程官网：http://www.runoob.com
```

函数参数名称

函数参数都有一个外部参数名和一个局部参数名。

局部参数名

局部参数名在函数的实现内部使用。

```
func sample(number: Int) {
   println(number)
}
```

以上实例中 number 为局部参数名，只能在函数体内使用。

```
import Cocoa

func sample(number: Int) {
   print(number)
}
sample(1)
sample(2)
sample(3)
```

以上程序执行输出结果为：

```
1
2
3
```

外部参数名

你可以在局部参数名前指定外部参数名，中间以空格分隔，外部参数名用于在函数调用时传递给函数的参数。

如下你可以定义以下两个函数参数名并调用它：

```
import Cocoa

func pow(firstArg a: Int, secondArg b: Int) -> Int {
   var res = a
   for _ in 1..<b {
      res = res * a
   }
   print(res)
   return res
}
pow(firstArg:5, secondArg:3)
```

以上程序执行输出结果为：

```
125
```

> 注意 如果你提供了外部参数名，那么函数在被调用时，必须使用外部参数名。

可变参数

可变参数可以接受零个或多个值。函数调用时，你可以用可变参数来指定函数参数，其数量是不确定的。

可变参数通过在变量类型名后面加入（...）的方式来定义。

```
import Cocoa

func vari<N>(members: N...){
    for i in members {
        print(i)
    }
}
vari(4,3,5)
vari(4.5, 3.1, 5.6)
vari("Google", "Baidu", "Runoob")
```

以上程序执行输出结果为：

```
4
3
5
4.5
3.1
5.6
Google
Baidu
Runoob
```

常量，变量及 I/O 参数

一般默认在函数中定义的参数都是常量参数，也就是这个参数你只可以查询使用，不能改变它的值。

如果想要声明一个变量参数，可以在前面加上var，这样就可以改变这个参数的值了。

例如：

```
func  getName(var id:String).........
```

此时这个id值可以在函数中改变。

一般默认的参数传递都是传值调用的，而不是传引用。 所以传入的参数在函数内改变，并不影响原来的那个参数。传入的只是这个参数的副本。

变量参数，正如上面所述，仅仅能在函数体内被更改。如果你想要一个函数可以修改参数的值，并且想要在这些修改在函数调用结束后仍然存在，那么就应该把这个参数定义为输入输出参数（In-Out Parameters）。

定义一个输入输出参数时，在参数定义前加 inout 关键字。一个输入输出参数有传入函数的值，这个值被函数修改，然后被传出函数，替换原来的值。

实例

```
import Cocoa

func swapTwoInts(var a:Int,var b:Int){

    let t = a
    a = b
    b = t
}

var x = 0,y = 100
print("x = \(x) ;y = \(y)")

swapTwoInts(x, b:y)
print("x = \(x) ;y = \(y)")
```

以上程序执行输出结果为：

```
x = 0 ;y = 100
x = 0 ;y = 100
```

修改方法是使用inout关键字：

```
import Cocoa

func swapTwoInts(inout a:Int,inout b:Int){

    let t = a
    a = b
    b = t
}

var x = 0,y = 100
print("x = \(x) ;y = \(y)")

swapTwoInts(&x, b:&y)
print("x = \(x) ;y = \(y)")
```

以上程序执行输出结果为：

```
x = 0 ;y = 100
x = 100 ;y = 0
```

函数类型及使用

每个函数都有种特定的函数类型，由函数的参数类型和返回类型组成。

```
func inputs(no1: Int, no2: Int) -> Int {
   return no1/no2
}
```

实例如下：

```
import Cocoa

func inputs(no1: Int, no2: Int) -> Int {
    return no1/no2
}
print(inputs(20,no2:10))
print(inputs(36,no2:6))
```

以上程序执行输出结果为：

```
2
6
```

以上函数定义了两个 Int 参数类型，返回值也为 Int 类型。

接下来我们看下如下函数，函数定义了参数为 String 类型，返回值为 String 类型。

```
Func inputstr(name: String) -> String {
    return name
}
```

函数也可以定义任何参数及类型，如下所示：

```
import Cocoa

func inputstr() {
    print("菜鸟教程")
    print("www.runoob.com")
}
inputstr()
```

以上程序执行输出结果为：

```
菜鸟教程
www.runoob.com
```

使用函数类型

在 Swift 中，使用函数类型就像使用其他类型一样。例如，你可以定义一个类型为函数的常量或变量，并将适当的函数赋值给它：

```
var addition: (Int, Int) -> Int = sum
```

解析:

"定义一个叫做 `addition` 的变量，参数与返回值类型均是 `Int` ，并让这个新变量指向 `sum` 函数"。

`sum` 和 `addition` 有同样的类型，所以以上操作是合法的。

现在，你可以用 addition 来调用被赋值的函数了：

```
import Cocoa

func sum(a: Int, b: Int) -> Int {
   return a + b
}
var addition: (Int, Int) -> Int = sum
print("输出结果: \(addition(40, 89))")
```

以上程序执行输出结果为：

```
输出结果: 129
```

函数类型作为参数类型、函数类型作为返回类型

我们可以将函数作为参数传递给另外一个参数：

```
import Cocoa

func sum(a: Int, b: Int) -> Int {
    return a + b
}
var addition: (Int, Int) -> Int = sum
print("输出结果: \(addition(40, 89))")

func another(addition: (Int, Int) -> Int, a: Int, b: Int) {
    print("输出结果: \(addition(a, b))")
}
another(sum, a: 10, b: 20)
```

以上程序执行输出结果为：

```
输出结果: 129
输出结果: 30
```

函数嵌套

函数嵌套指的是函数内定义一个新的函数，外部的函数可以调用函数内定义的函数。

实例如下：

```
import Cocoa

func calcDecrement(forDecrement total: Int) -> () -> Int {
   var overallDecrement = 0
   func decrementer() -> Int {
      overallDecrement -= total
      return overallDecrement
   }
   return decrementer
}
let decrem = calcDecrement(forDecrement: 30)
print(decrem())
```

以上程序执行输出结果为：

```
-30
```

Swift 闭包

闭包(Closures)是自包含的功能代码块，可以在代码中使用或者用来作为参数传值。

Swift 中的闭包与 C 和 Objective-C 中的代码块（blocks）以及其他一些编程语言中的 匿名函数比较相似。

全局函数和嵌套函数其实就是特殊的闭包。

闭包的形式有：

全局函数	嵌套函数	闭包表达式
有名字但不能捕获任何值。	有名字，也能捕获封闭函数内的值。	无名闭包，使用轻量级语法，可以根据上下文环境捕获值。

Swift中的闭包有很多优化的地方:

1. 根据上下文推断参数和返回值类型
2. 从单行表达式闭包中隐式返回（也就是闭包体只有一行代码，可以省略return）
3. 可以使用简化参数名，如$0, $1(从0开始，表示第i个参数...)
4. 提供了尾随闭包语法(Trailing closure syntax)

语法

以下定义了一个接收参数并返回指定类型的闭包语法：

```
{(parameters) -> return type in
   statements
}
```

实例

```
import Cocoa

let studname = { print("Swift 闭包实例。") }
studname()
```

以上程序执行输出结果为：

```
Swift 闭包实例。
```

以下闭包形式接收两个参数并返回布尔值：

```
{(Int, Int) -> Bool in
   Statement1
   Statement 2
    ---
   Statement n
}
```

实例

```
import Cocoa

let divide = {(val1: Int, val2: Int) -> Int in
   return val1 / val2
}
let result = divide(200, 20)
print (result)
```

以上程序执行输出结果为：

```
10
```

闭包表达式

闭包表达式是一种利用简洁语法构建内联闭包的方式。 闭包表达式提供了一些语法优化，使得撰写闭包变得简单明了。

sort 函数

Swift 标准库提供了名为sort的函数，会根据您提供的用于排序的闭包函数将已知类型数组中的值进行排序。

排序完成后，sort(*:)方法会返回一个与原数组大小相同,包含同类型元素且元素已正确排序的新数组，原数组不会被sort(*:)方法修改。

sort(_:)方法需要传入两个参数：

- 已知类型的数组
- 闭包函数，该闭包函数需要传入与数组元素类型相同的两个值，并返回一个布尔类型值来表明当排序结束后传入的第一个参数排在第二个参数前面还是后面。如果第一个参数值出现在第二个参数值前面，排序闭包函数需要返回 `true` ，反之返回 `false` 。

实例

```
import Cocoa

let names = ["AT", "AE", "D", "S", "BE"]

// 使用普通函数(或内嵌函数)提供排序功能,闭包函数类型需为(String, String) ->
func backwards(s1: String, s2: String) -> Bool {
    return s1 > s2
}
var reversed = names.sort(backwards)

print(reversed)
```

以上程序执行输出结果为：

```
["S", "D", "BE", "AT", "AE"]
```

如果第一个字符串 (s1) 大于第二个字符串 (s2)，backwards函数返回true，表示在新的数组中s1应该出现在s2前。 对于字符串中的字符来说，"大于" 表示 "按照字母顺序较晚出现"。 这意味着字母"B"大于字母"A"，字符串"S"大于字符串"D"。 其将进行字母逆序排序，"AT"将会排在"AE"之前。

参数名称缩写

Swift 自动为内联函数提供了参数名称缩写功能，您可以直接通过$0,$1,$2来顺序调用闭包的参数。

实例

```
import Cocoa

let names = ["AT", "AE", "D", "S", "BE"]

var reversed = names.sort( { $0 > $1 } )
print(reversed)
```

$0和$1表示闭包中第一个和第二个String类型的参数。

以上程序执行输出结果为：

```
["S", "D", "BE", "AT", "AE"]
```

如果你在闭包表达式中使用参数名称缩写, 您可以在闭包参数列表中省略对其定义, 并且对应参数名称缩写的类型会通过函数类型进行推断。in 关键字同样也可以被省略.

运算符函数

实际上还有一种更简短的方式来撰写上面例子中的闭包表达式。

Swift 的 `String` 类型定义了关于大于号 (`>`) 的字符串实现，其作为一个函数接受两个 `String` 类型的参数并返回 `Bool` 类型的值。 而这正好与 `sort(_:)` 方法的第二个参数需要的函数类型相符合。 因此，您可以简单地传递一个大于号，Swift可以自动推断出您想使用大于号的字符串函数实现：

```
import Cocoa

let names = ["AT", "AE", "D", "S", "BE"]

var reversed = names.sort(>)
print(reversed)
```

以上程序执行输出结果为：

```
["S", "D", "BE", "AT", "AE"]
```

尾随闭包

尾随闭包是一个书写在函数括号之后的闭包表达式，函数支持将其作为最后一个参数调用。

```
func someFunctionThatTakesAClosure(closure: () -> Void) { //
```

实例

```
func someFunctionThatTakesAClosure(closure: () -> Void) {
    // 函数体部分
}

// 以下是不使用尾随闭包进行函数调用
someFunctionThatTakesAClosure({
    // 闭包主体部分
})

// 以下是使用尾随闭包进行函数调用
someFunctionThatTakesAClosure() {
  // 闭包主体部分
}
```

sort() 后的 { \$0 > \$1} 为尾随闭包。

以上程序执行输出结果为：

```
["S", "D", "BE", "AT", "AE"]
```

> 注意： 如果函数只需要闭包表达式一个参数，当您使用尾随闭包时，您甚至可以把 `()` 省略掉。
>
> ```
> reversed = names.sort { $0 > $1 }
> ```

捕获值

闭包可以在其定义的上下文中捕获常量或变量。

即使定义这些常量和变量的原域已经不存在，闭包仍然可以在闭包函数体内引用和修改这些值。

Swift最简单的闭包形式是嵌套函数，也就是定义在其他函数的函数体内的函数。

嵌套函数可以捕获其外部函数所有的参数以及定义的常量和变量。

看这个例子：

```
func makeIncrementor(forIncrement amount: Int) -> () -> Int {
    var runningTotal = 0
    func incrementor() -> Int {
        runningTotal += amount
        return runningTotal
    }
    return incrementor
}
```

一个函数makeIncrementor，它有一个Int型的参数amout, 并且它有一个外部参数名字forIncremet，意味着你调用的时候，必须使用这个外部名字。返回值是一个 `()-> Int` 的函数。

函数题内，声明了变量runningTotal 和一个函数incrementor。

incrementor函数并没有获取任何参数，但是在函数体内访问了runningTotal和amount变量。这是因为其通过捕获在包含它的函数体内已经存在的runningTotal和amount变量而实现。

由于没有修改amount变量，incrementor实际上捕获并存储了该变量的一个副本，而该副本随着incrementor一同被存储。

所以我们调用这个函数时会累加：

```
import Cocoa

func makeIncrementor(forIncrement amount: Int) -> () -> Int {
    var runningTotal = 0
    func incrementor() -> Int {
        runningTotal += amount
        return runningTotal
    }
    return incrementor
}

let incrementByTen = makeIncrementor(forIncrement: 10)

// 返回的值为10
print(incrementByTen())

// 返回的值为20
print(incrementByTen())

// 返回的值为30
print(incrementByTen())
```

以上程序执行输出结果为：

```
10
20
30
```

闭包是引用类型

上面的例子中，incrementByTen是常量，但是这些常量指向的闭包仍然可以增加其捕获的变量值。

这是因为函数和闭包都是引用类型。

无论您将函数/闭包赋值给一个常量还是变量，您实际上都是将常量/变量的值设置为对应函数/闭包的引用。 上面的例子中，incrementByTen指向闭包的引用是一个常量，而并非闭包内容本身。

这也意味着如果您将闭包赋值给了两个不同的常量/变量，两个值都会指向同一个闭包：

```
import Cocoa

func makeIncrementor(forIncrement amount: Int) -> () -> Int {
    var runningTotal = 0
    func incrementor() -> Int {
        runningTotal += amount
        return runningTotal
    }
    return incrementor
}

let incrementByTen = makeIncrementor(forIncrement: 10)

// 返回的值为10
incrementByTen()

// 返回的值为20
incrementByTen()

// 返回的值为30
incrementByTen()

// 返回的值为40
incrementByTen()

let alsoIncrementByTen = incrementByTen

// 返回的值也为50
print(alsoIncrementByTen())
```

以上程序执行输出结果为：

```
50
```

Swift 枚举

枚举简单的说也是一种数据类型，只不过是这种数据类型只包含自定义的特定数据，它是一组有共同特性的数据的集合。

Swift 的枚举类似于 Objective C 和 C 的结构，枚举的功能为:

- 它声明在类中，可以通过实例化类来访问它的值。
- 枚举也可以定义构造函数（initializers）来提供一个初始成员值；可以在原始的实现基础上扩展它们的功能。
- 可以遵守协议（protocols）来提供标准的功能。

语法

Swift 中使用 enum 关键词来创建枚举并且把它们的整个定义放在一对大括号内：

```
enum enumname {
   // 枚举定义放在这里
}
```

例如我们定义以下表示星期的枚举：

```
import Cocoa

// 定义枚举
enum DaysofaWeek {
    case Sunday
    case Monday
    case TUESDAY
    case WEDNESDAY
    case THURSDAY
    case FRIDAY
    case Saturday
}

var weekDay = DaysofaWeek.THURSDAY
weekDay = .THURSDAY
switch weekDay
{
case .Sunday:
    print("星期天")
case .Monday:
    print("星期一")
case .TUESDAY:
    print("星期二")
case .WEDNESDAY:
    print("星期三")
case .THURSDAY:
    print("星期四")
case .FRIDAY:
    print("星期五")
case .Saturday:
    print("星期六")
}
```

以上程序执行输出结果为：

```
星期四
```

枚举中定义的值（如 `Sunday`，`Monday`，…… 和 `Saturday`）是这个枚举的成员值（或成员）。`case` 关键词表示一行新的成员值将被定义。

> 注意： 和 C 和 Objective-C 不同，Swift 的枚举成员在被创建时不会被赋予一个默认的整型值。在上面的 `DaysofaWeek` 例子中，`Sunday`，`Monday`，…… 和 `Saturday` 不会隐式地赋值为 `0`，`1`，…… 和 `6`。相反，这些枚举成员本身就有完备的值，这些值是已经明确定义好的 `DaysofaWeek` 类型。

```
var weekDay = DaysofaWeek.THURSDAY
```

`weekDay` 的类型可以在它被 `DaysofaWeek` 的一个可能值初始化时推断出来。一旦 `weekDay` 被声明为一个 `DaysofaWeek` ，你可以使用一个缩写语法（.）将其设置为另一个 `DaysofaWeek` 的值：

```
var weekDay = .THURSDAY
```

当 `weekDay` 的类型已知时，再次为其赋值可以省略枚举名。使用显式类型的枚举值可以让代码具有更好的可读性。

枚举可分为相关值与原始值。

相关值与原始值的区别

相关值	原始值
不同数据类型	相同数据类型
实例: enum {10,0.8,"Hello"}	实例: enum {10,35,50}
值的创建基于常量或变量	预先填充的值
相关值是当你在创建一个基于枚举成员的新常量或变量时才会被设置，并且每次当你这么做得时候，它的值可以是不同的。	原始值始终是相同的

相关值

以下实例中我们定义一个名为 Student 的枚举类型，它可以是 Name 的一个相关值（Int，Int，Int，Int），或者是 Mark 的一个字符串类型（String）相关值。

```
import Cocoa

enum Student{
    case Name(String)
    case Mark(Int,Int,Int)
}
var studDetails = Student.Name("Runoob")
var studMarks = Student.Mark(98,97,95)
switch studMarks {
case .Name(let studName):
    print("学生的名字是: \(studName)。")
case .Mark(let Mark1, let Mark2, let Mark3):
    print("学生的成绩是: \(Mark1),\(Mark2),\(Mark3)。")
}
```

以上程序执行输出结果为：

```
学生的成绩是: 98,97,95。
```

原始值

原始值可以是字符串，字符，或者任何整型值或浮点型值。每个原始值在它的枚举声明中必须是唯一的。

在原始值为整数的枚举时，不需要显式的为每一个成员赋值，Swift会自动为你赋值。

```
import Cocoa

enum Month: Int {
    case January = 1, February, March, April, May, June, July, Augu
}

let yearMonth = Month.May.rawValue
print("数字月份为: \(yearMonth)。")
```

以上程序执行输出结果为：

```
数字月份为: 5。
```

Swift 结构体

Swift 结构体是构建代码所用的一种通用且灵活的构造体。

我们可以为结构体定义属性（常量、变量）和添加方法，从而扩展结构体的功能。

与 C 和 Objective C 不同的是：

- 结构体不需要包含实现文件和接口。
- 结构体允许我们创建一个单一文件，且系统会自动生成面向其它代码的外部接口。

结构体总是通过被复制的方式在代码中传递，因此它的值是不可修改的。

语法

我们通过关键字 struct 来定义结构体：

```
struct nameStruct {
   Definition 1
   Definition 2
   ……
   Definition N
}
```

实例

我们定义一个名为 MarkStruct 的结构体 ，结构体的属性为学生三个科目的分数，数据类型为 Int：

```
struct MarkStruct{
   var mark1: Int
   var mark2: Int
   var mark3: Int
}
```

我们可以通过结构体名来访问结构体成员。

结构体实例化使用 **let** 关键字：

```
import Cocoa

struct studentMarks {
   var mark1 = 100
   var mark2 = 78
   var mark3 = 98
}
let marks = studentMarks()
print("Mark1 是 \(marks.mark1)")
print("Mark2 是 \(marks.mark2)")
print("Mark3 是 \(marks.mark3)")
```

以上程序执行输出结果为：

```
Mark1 是 100
Mark2 是 78
Mark3 是 98
```

实例中，我们通过结构体名 'studentMarks' 访问学生的成绩。结构体成员初始化为 mark1, mark2, mark3，数据类型为整型。

然后我们通过使用 **let** 关键字将结构体 studentMarks() 实例化并传递给 marks。

最后我们就通过 **.** 号来访问结构体成员的值。

以下实例化通过结构体实例化时传值并克隆一个结构体：

```
import Cocoa

struct MarksStruct {
   var mark: Int

   init(mark: Int) {
      self.mark = mark
   }
}
var aStruct = MarksStruct(mark: 98)
var bStruct = aStruct // aStruct 和 bStruct 是使用相同值的结构体！
bStruct.mark = 97
print(aStruct.mark) // 98
print(bStruct.mark) // 97
```

以上程序执行输出结果为：

```
98
97
```

结构体应用

在你的代码中，你可以使用结构体来定义你的自定义数据类型。

结构体实例总是通过值传递来定义你的自定义数据类型。

按照通用的准则，当符合一条或多条以下条件时，请考虑构建结构体：

- 结构体的主要目的是用来封装少量相关简单数据值。
- 有理由预计一个结构体实例在赋值或传递时，封装的数据将会被拷贝而不是被引用。
- 任何在结构体中储存的值类型属性，也将会被拷贝，而不是被引用。
- 结构体不需要去继承另一个已存在类型的属性或者行为。

举例来说，以下情境中适合使用结构体：

- 几何形状的大小，封装一个 `width` 属性和 `height` 属性，两者均为 `Double` 类型。
- 一定范围内的路径，封装一个 `start` 属性和 `length` 属性，两者均为 `Int` 类型。
- 三维坐标系内一点，封装 `x` ，`y` 和 `z` 属性，三者均为 `Double` 类型。

结构体实例是通过值传递而不是通过引用传递。

```
import Cocoa

struct markStruct{
    var mark1: Int
    var mark2: Int
    var mark3: Int

    init(mark1: Int, mark2: Int, mark3: Int){
        self.mark1 = mark1
        self.mark2 = mark2
        self.mark3 = mark3
    }
}

print("优异成绩:")
var marks = markStruct(mark1: 98, mark2: 96, mark3:100)
print(marks.mark1)
print(marks.mark2)
print(marks.mark3)

print("糟糕成绩:")
var fail = markStruct(mark1: 34, mark2: 42, mark3: 13)
print(fail.mark1)
print(fail.mark2)
print(fail.mark3)
```

以上程序执行输出结果为：

```
优异成绩:
98
96
100
糟糕成绩:
34
42
13
```

以上实例中我们定义了结构体 markStruct，三个成员属性：mark1, mark2 和 mark3。结构体内使用成员属性使用 self 关键字。

从实例中我们可以很好的理解到结构体实例是通过值传递的。

Swift 类

Swift 类是构建代码所用的一种通用且灵活的构造体。

我们可以为类定义属性（常量、变量）和方法。

与其他编程语言所不同的是，Swift 并不要求你为自定义类去创建独立的接口和实现文件。你所要做的是在一个单一文件中定义一个类，系统会自动生成面向其它代码的外部接口。

类和结构体对比

Swift 中类和结构体有很多共同点。共同处在于：

- 定义属性用于存储值
- 定义方法用于提供功能
- 定义附属脚本用于访问值
- 定义构造器用于生成初始化值
- 通过扩展以增加默认实现的功能
- 符合协议以对某类提供标准功能

与结构体相比，类还有如下的附加功能：

- 继承允许一个类继承另一个类的特征
- 类型转换允许在运行时检查和解释一个类实例的类型
- 解构器允许一个类实例释放任何其所被分配的资源
- 引用计数允许对一个类的多次引用

语法:

```
Class classname {
   Definition 1
   Definition 2
   ……
   Definition N
}
```

类定义

```
class student{
   var studname: String
   var mark: Int
   var mark2: Int
}
```

实例化类：

```
let studrecord = student()
```

实例

```
import Cocoa

class MarksStruct {
    var mark: Int
    init(mark: Int) {
        self.mark = mark
    }
}

class studentMarks {
    var mark = 300
}
let marks = studentMarks()
print("成绩为 \(marks.mark)")
```

以上程序执行输出结果为：

```
成绩为  300
```

作为引用类型访问类属性

类的属性可以通过 . 来访问。格式为：实例化类名.属性名：

```
import Cocoa

class MarksStruct {
   var mark: Int
   init(mark: Int) {
      self.mark = mark
   }
}

class studentMarks {
   var mark1 = 300
   var mark2 = 400
   var mark3 = 900
}
let marks = studentMarks()
print("Mark1 is \(marks.mark1)")
print("Mark2 is \(marks.mark2)")
print("Mark3 is \(marks.mark3)")
```

以上程序执行输出结果为：

```
Mark1 is 300
Mark2 is 400
Mark3 is 900
```

恒等运算符

因为类是引用类型，有可能有多个常量和变量在后台同时引用某一个类实例。

为了能够判定两个常量或者变量是否引用同一个类实例，Swift 内建了两个恒等运算符：

恒等运算符	不恒等运算符
运算符为：===	运算符为：!==
如果两个常量或者变量引用同一个类实例则返回 true	如果两个常量或者变量引用不同一个类实例则返回 true

实例

```
import Cocoa

class SampleClass: Equatable {
    let myProperty: String
    init(s: String) {
        myProperty = s
    }
}
func ==(lhs: SampleClass, rhs: SampleClass) -> Bool {
    return lhs.myProperty == rhs.myProperty
}

let spClass1 = SampleClass(s: "Hello")
let spClass2 = SampleClass(s: "Hello")

if spClass1 === spClass2 {// false
    print("引用相同的类实例 \(spClass1)")
}

if spClass1 !== spClass2 {// true
    print("引用不相同的类实例 \(spClass2)")
}
```

以上程序执行输出结果为：

```
引用不相同的类实例  SampleClass
```

Swift 属性

Swift 属性将值跟特定的类、结构或枚举关联。

属性可分为存储属性和计算属性:

存储属性	计算属性
存储常量或变量作为实例的一部分	计算（而不是存储）一个值
用于类和结构体	用于类、结构体和枚举

存储属性和计算属性通常用于特定类型的实例。

属性也可以直接用于类型本身，这种属性称为类型属性。

另外，还可以定义属性观察器来监控属性值的变化，以此来触发一个自定义的操作。属性观察器可以添加到自己写的存储属性上，也可以添加到从父类继承的属性上。

存储属性

简单来说，一个存储属性就是存储在特定类或结构体的实例里的一个常量或变量。

存储属性可以是变量存储属性（用关键字var定义），也可以是常量存储属性（用关键字let定义）。

- 可以在定义存储属性的时候指定默认值
- 也可以在构造过程中设置或修改存储属性的值，甚至修改常量存储属性的值

```
import Cocoa

struct Number
{
   var digits: Int
   let pi = 3.1415
}

var n = Number(digits: 12345)
n.digits = 67

print("\(n.digits)")
print("\(n.pi)")
```

以上程序执行输出结果为：

```
67
3.1415
```

考虑以下代码：

```
let pi = 3.1415
```

代码中 pi 在定义存储属性的时候指定默认值（pi = 3.1415），所以不管你什么时候实例化结构体，它都不会改变。

如果你定义的是一个常量存储属性，如果尝试修改它就会报错，如下所示：

```
import Cocoa

struct Number
{
    var digits: Int
    let numbers = 3.1415
}

var n = Number(digits: 12345)
n.digits = 67

print("\(n.digits)")
print("\(n.numbers)")
n.numbers = 8.7
```

以上程序，执行会报错，错误如下所示：

```
error: cannot assign to property: 'numbers' is a 'let' constant
n.numbers = 8.7
```

意思为 'numbers' 是一个常量，你能修改它。

延迟存储属性

延迟存储属性是指当第一次被调用的时候才会计算其初始值的属性。

在属性声明前使用 **lazy** 来标示一个延迟存储属性。

> 注意：
>
> 必须将延迟存储属性声明成变量（使用 `var` 关键字），因为属性的值在实例构造完成之前可能无法得到。而常量属性在构造过程完成之前必须要有初始值，因此无法声明成延迟属性。

延迟存储属性一般用于：

- 延迟对象的创建。
- 当属性的值依赖于其他未知类

```
import Cocoa

class sample {
    lazy var no = number() // `var` 关键字是必须的
}

class number {
    var name = "Runoob Swift 教程"
}

var firstsample = sample()
print(firstsample.no.name)
```

以上程序执行输出结果为：

```
Runoob Swift 教程
```

实例化变量

如果您有过 Objective-C 经验，应该知道Objective-C 为类实例存储值和引用提供两种方法。对于属性来说，也可以使用实例变量作为属性值的后端存储。

Swift 编程语言中把这些理论统一用属性来实现。Swift 中的属性没有对应的实例变量，属性的后端存储也无法直接访问。这就避免了不同场景下访问方式的困扰，同时也将属性的定义简化成一个语句。

一个类型中属性的全部信息——包括命名、类型和内存管理特征——都在唯一一个地方（类型定义中）定义。

计算属性

除存储属性外，类、结构体和枚举可以定义计算属性，计算属性不直接存储值，而是提供一个 getter 来获取值，一个可选的 setter 来间接设置其他属性或变量的值。

```
import Cocoa

class sample {
    var no1 = 0.0, no2 = 0.0
    var length = 300.0, breadth = 150.0

    var middle: (Double, Double) {
        get{
            return (length / 2, breadth / 2)
        }
        set(axis){
            no1 = axis.0 - (length / 2)
            no2 = axis.1 - (breadth / 2)
        }
    }
}

var result = sample()
print(result.middle)
result.middle = (0.0, 10.0)

print(result.no1)
print(result.no2)
```

以上程序执行输出结果为：

```
(150.0, 75.0)
-150.0
-65.0
```

如果计算属性的 setter 没有定义表示新值的参数名，则可以使用默认名称 newValue。

只读计算属性

只有 getter 没有 setter 的计算属性就是只读计算属性。

只读计算属性总是返回一个值，可以通过点(.)运算符访问，但不能设置新的值。

```
import Cocoa

class film {
    var head = ""
    var duration = 0.0
    var metaInfo: [String:String] {
        return [
            "head": self.head,
            "duration":"\(self.duration)"
        ]
    }
}

var movie = film()
movie.head = "Swift 属性"
movie.duration = 3.09

print(movie.metaInfo["head"]!)
print(movie.metaInfo["duration"]!)
```

以上程序执行输出结果为：

```
Swift 属性
3.09
```

> 注意：
>
> 必须使用 `var` 关键字定义计算属性，包括只读计算属性，因为它们的值不是固定的。 `let` 关键字只用来声明常量属性，表示初始化后再也无法修改的值。

属性观察器

属性观察器监控和响应属性值的变化，每次属性被设置值的时候都会调用属性观察器，甚至新的值和现在的值相同的时候也不例外。

可以为除了延迟存储属性之外的其他存储属性添加属性观察器，也可以通过重载属性的方式为继承的属性（包括存储属性和计算属性）添加属性观察器。

> 注意：
>
> 不需要为无法重载的计算属性添加属性观察器，因为可以通过 setter 直接监控和响应值的变化。

可以为属性添加如下的一个或全部观察器：

- `willSet` 在设置新的值之前调用
- `didSet` 在新的值被设置之后立即调用

- willSet和didSet观察器在属性初始化过程中不会被调用

```
import Cocoa

class Samplepgm {
    var counter: Int = 0{
        willSet(newTotal){
            print("计数器: \(newTotal)")
        }
        didSet{
            if counter > oldValue {
                print("新增数 \(counter - oldValue)")
            }
        }
    }
}
let NewCounter = Samplepgm()
NewCounter.counter = 100
NewCounter.counter = 800
```

以上程序执行输出结果为：

```
计数器: 100
新增数 100
计数器: 800
新增数 700
```

全局变量和局部变量

计算属性和属性观察器所描述的模式也可以用于全局变量和局部变量。

局部变量	全局变量
在函数、方法或闭包内部定义的变量。	函数、方法、闭包或任何类型之外定义的变量。
用于存储和检索值。	用于存储和检索值。
存储属性用于获取和设置值。	存储属性用于获取和设置值。
也用于计算属性。	也用于计算属性。

类型属性

类型属性是作为类型定义的一部分写在类型最外层的花括号（{}）内。

使用关键字 static 来定义值类型的类型属性，关键字 class 来为类定义类型属性。

```
struct Structname {
   static var storedTypeProperty = " "
   static var computedTypeProperty: Int {
      // 这里返回一个 Int 值
   }
}

enum Enumname {
   static var storedTypeProperty = " "
   static var computedTypeProperty: Int {
      // 这里返回一个 Int 值
   }
}

class Classname {
   class var computedTypeProperty: Int {
      // 这里返回一个 Int 值
   }
}
```

> 注意：
>
> 例子中的计算型类型属性是只读的，但也可以定义可读可写的计算型类型属性，跟实例计算属性的语法类似。

获取和设置类型属性的值

类似于实例的属性，类型属性的访问也是通过点运算符(.)来进行。但是，类型属性是通过类型本身来获取和设置，而不是通过实例。实例如下：

```
import Cocoa

struct StudMarks {
   static let markCount = 97
   static var totalCount = 0
   var InternalMarks: Int = 0 {
      didSet {
         if InternalMarks > StudMarks.markCount {
            InternalMarks = StudMarks.markCount
         }
         if InternalMarks > StudMarks.totalCount {
            StudMarks.totalCount = InternalMarks
         }
      }
   }
}

var stud1Mark1 = StudMarks()
var stud1Mark2 = StudMarks()

stud1Mark1.InternalMarks = 98
print(stud1Mark1.InternalMarks)

stud1Mark2.InternalMarks = 87
print(stud1Mark2.InternalMarks)
```

以上程序执行输出结果为：

```
97
87
```

Swift 方法

Swift 方法是与某些特定类型相关联的函数

在 Objective-C 中，类是唯一能定义方法的类型。但在 Swift 中，你不仅能选择是否要定义一个类/结构体/枚举，还能灵活的在你创建的类型（类/结构体/枚举）上定义方法。

实例方法

在 Swift 语言中，实例方法是属于某个特定类、结构体或者枚举类型实例的方法。

实例方法提供以下方法：

- 可以访问和修改实例属性
- 提供与实例目的相关的功能

实例方法要写在它所属的类型的前后大括号({})之间。

实例方法能够隐式访问它所属类型的所有的其他实例方法和属性。

实例方法只能被它所属的类的某个特定实例调用。

实例方法不能脱离于现存的实例而被调用。

语法

```
func funcname(Parameters) -> returntype
{
    Statement1
    Statement2
    ……
    Statement N
    return parameters
}
```

实例

```
import Cocoa

class Counter {
    var count = 0
    func increment() {
        count++
    }
    func incrementBy(amount: Int) {
        count += amount
    }
    func reset() {
        count = 0
    }
}
// 初始计数值是0
let counter = Counter()

// 计数值现在是1
counter.increment()

// 计数值现在是6
counter.incrementBy(5)
print(counter.count)

// 计数值现在是0
counter.reset()
print(counter.count)
```

以上程序执行输出结果为：

```
6
0
```

Counter类定义了三个实例方法：

- `increment` 让计数器按一递增；
- `incrementBy(amount: Int)` 让计数器按一个指定的整数值递增；
- `reset` 将计数器重置为0。

`Counter` 这个类还声明了一个可变属性 `count` ，用它来保持对当前计数器值的追踪。

方法的局部参数名称和外部参数名称

Swift 函数参数可以同时有一个局部名称（在函数体内部使用）和一个外部名称（在调用函数时使用

Swift 中的方法和 Objective-C 中的方法极其相似。像在 Objective-C 中一样，Swift 中方法的名称通常用一个介词指向方法的第一个参数，比如：with，for，by等等。

Swift 默认仅给方法的第一个参数名称一个局部参数名称;默认同时给第二个和后续的参数名称为全局参数名称。

以下实例中 'no1' 在swift中声明为局部参数名称。'no2' 用于全局的声明并通过外部程序访问。

```
import Cocoa

class division {
    var count: Int = 0
    func incrementBy(no1: Int, no2: Int) {
        count = no1 / no2
        print(count)
    }
}

let counter = division()
counter.incrementBy(1800, no2: 3)
counter.incrementBy(1600, no2: 5)
counter.incrementBy(11000, no2: 3)
```

以上程序执行输出结果为：

```
600
320
3666
```

是否提供外部名称设置

我们强制在第一个参数添加外部名称把这个局部名称当作外部名称使用（Swift 2.0 前是使用 # 号）。

```
import Cocoa

class multiplication {
    var count: Int = 0
    func incrementBy(first no1: Int, no2: Int) {
        count = no1 * no2
        print(count)
    }
}

let counter = multiplication()
counter.incrementBy(first: 800, no2: 3)
counter.incrementBy(first: 100, no2: 5)
counter.incrementBy(first: 15000, no2: 3)
```

以上程序执行输出结果为：

```
2400
500
45000
```

self 属性

类型的每一个实例都有一个隐含属性叫做self，self 完全等同于该实例本身。

你可以在一个实例的实例方法中使用这个隐含的self属性来引用当前实例。

```
import Cocoa

class calculations {
    let a: Int
    let b: Int
    let res: Int

    init(a: Int, b: Int) {
        self.a = a
        self.b = b
        res = a + b
        print("Self 内: \(res)")
    }

    func tot(c: Int) -> Int {
        return res - c
    }

    func result() {
        print("结果为: \(tot(20))")
        print("结果为: \(tot(50))")
    }
}

let pri = calculations(a: 600, b: 300)
let sum = calculations(a: 1200, b: 300)

pri.result()
sum.result()
```

以上程序执行输出结果为：

```
Self 内: 900
Self 内: 1500
结果为: 880
结果为: 850
结果为: 1480
结果为: 1450
```

在实例方法中修改值类型

Swift 语言中结构体和枚举是值类型。一般情况下，值类型的属性不能在它的实例方法中被修改。

但是，如果你确实需要在某个具体的方法中修改结构体或者枚举的属性，你可以选择变异(mutating)这个方法，然后方法就可以从方法内部改变它的属性；并且它做的任何改变在方法结束时还会保留在原始结构中。

方法还可以给它隐含的self属性赋值一个全新的实例，这个新实例在方法结束后将替换原来的实例。

```
import Cocoa

struct area {
    var length = 1
    var breadth = 1

    func area() -> Int {
        return length * breadth
    }

    mutating func scaleBy(res: Int) {
        length *= res
        breadth *= res

        print(length)
        print(breadth)
    }
}

var val = area(length: 3, breadth: 5)
val.scaleBy(3)
val.scaleBy(30)
val.scaleBy(300)
```

以上程序执行输出结果为：

```
9
15
270
450
81000
135000
```

在可变方法中给 self 赋值

可变方法能够赋给隐含属性 self 一个全新的实例。

```
import Cocoa

struct area {
    var length = 1
    var breadth = 1

    func area() -> Int {
        return length * breadth
    }

    mutating func scaleBy(res: Int) {
        self.length *= res
        self.breadth *= res
        print(length)
        print(breadth)
    }
}
var val = area(length: 3, breadth: 5)
val.scaleBy(13)
```

以上程序执行输出结果为：

```
39
65
```

类型方法

实例方法是被类型的某个实例调用的方法，你也可以定义类型本身调用的方法，这种方法就叫做类型方法。

声明结构体和枚举的类型方法，在方法的func关键字之前加上关键字static。类可能会用关键字class来允许子类重写父类的实现方法。

类型方法和实例方法一样用点号(.)语法调用。

```
import Cocoa

class Math
{
    class func abs(number: Int) -> Int
    {
        if number < 0
        {
            return (-number)
        }
        else
        {
            return number
        }
    }
}

struct absno
{
    static func abs(number: Int) -> Int
    {
        if number < 0
        {
            return (-number)
        }
        else
        {
            return number
        }
    }
}

let no = Math.abs(-35)
let num = absno.abs(-5)

print(no)
print(num)
```

以上程序执行输出结果为：

```
35
5
```

Swift 下标脚本

下标脚本 可以定义在类（Class）、结构体（structure）和枚举（enumeration）这些目标中，可以认为是访问对象、集合或序列的快捷方式，不需要再调用实例的特定的赋值和访问方法。

举例来说，用下标脚本访问一个数组(Array)实例中的元素可以这样写 someArray[index]，访问字典(Dictionary)实例中的元素可以这样写 someDictionary[key]。

对于同一个目标可以定义多个下标脚本，通过索引值类型的不同来进行重载，而且索引值的个数可以是多个。

下标脚本语法及应用

语法

下标脚本允许你通过在实例后面的方括号中传入一个或者多个的索引值来对实例进行访问和赋值。

语法类似于实例方法和计算型属性的混合。

与定义实例方法类似，定义下标脚本使用subscript关键字，显式声明入参（一个或多个）和返回类型。

与实例方法不同的是下标脚本可以设定为读写或只读。这种方式又有点像计算型属性的getter和setter：

```
subscript(index: Int) -> Int {
    get {
        // 用于下标脚本值的声明
    }
    set(newValue) {
        // 执行赋值操作
    }
}
```

实例 1

```
import Cocoa

struct subexample {
    let decrementer: Int
    subscript(index: Int) -> Int {
        return decrementer / index
    }
}
let division = subexample(decrementer: 100)

print("100 除以 9 等于 \(division[9])")
print("100 除以 2 等于 \(division[2])")
print("100 除以 3 等于 \(division[3])")
print("100 除以 5 等于 \(division[5])")
print("100 除以 7 等于 \(division[7])")
```

以上程序执行输出结果为：

```
100 除以 9 等于 11
100 除以 2 等于 50
100 除以 3 等于 33
100 除以 5 等于 20
100 除以 7 等于 14
```

在上例中，通过 subexample 结构体创建了一个除法运算的实例。数值 100 作为结构体构造函数传入参数初始化实例成员 decrementer。

你可以通过下标脚本来得到结果，比如 division[2] 即为 100 除以 2。

实例 2

```
import Cocoa

class daysofaweek {
    private var days = ["Sunday", "Monday", "Tuesday", "Wednesday",
        "Thursday", "Friday", "saturday"]
    subscript(index: Int) -> String {
        get {
            return days[index]   // 声明下标脚本的值
        }
        set(newValue) {
            self.days[index] = newValue   // 执行赋值操作
        }
    }
}
var p = daysofaweek()

print(p[0])
print(p[1])
print(p[2])
print(p[3])
```

以上程序执行输出结果为：

```
Sunday
Monday
Tuesday
Wednesday
```

用法

根据使用场景不同下标脚本也具有不同的含义。

通常下标脚本是用来访问集合（collection），列表（list）或序列（sequence）中元素的快捷方式。

你可以在你自己特定的类或结构体中自由的实现下标脚本来提供合适的功能。

例如，Swift 的字典（Dictionary）实现了通过下标脚本对其实例中存放的值进行存取操作。在下标脚本中使用和字典索引相同类型的值，并且把一个字典值类型的值赋值给这个下标脚来为字典设值：

```
import Cocoa

var numberOfLegs = ["spider": 8, "ant": 6, "cat": 4]
numberOfLegs["bird"] = 2

print(numberOfLegs)
```

以上程序执行输出结果为：

```
["ant": 6, "bird": 2, "cat": 4, "spider": 8]
```

上例定义一个名为numberOfLegs的变量并用一个字典字面量初始化出了包含三对键值的字典实例。numberOfLegs的字典存放值类型推断为Dictionary<string, int="">。字典实例创建完成之后通过下标脚本的方式将整型值2赋值到字典实例的索引为bird的位置中。</string,>

下标脚本选项

下标脚本允许任意数量的入参索引，并且每个入参类型也没有限制。

下标脚本的返回值也可以是任何类型。

下标脚本可以使用变量参数和可变参数。

一个类或结构体可以根据自身需要提供多个下标脚本实现，在定义下标脚本时通过传入参数的类型进行区分，使用下标脚本时会自动匹配合适的下标脚本实现运行，这就是下标脚本的重载。

```
import Cocoa

struct Matrix {
    let rows: Int, columns: Int
    var print: [Double]
    init(rows: Int, columns: Int) {
        self.rows = rows
        self.columns = columns
        print = Array(count: rows * columns, repeatedValue: 0.0)
    }
    subscript(row: Int, column: Int) -> Double {
        get {
            return print[(row * columns) + column]
        }
        set {
            print[(row * columns) + column] = newValue
        }
    }
}
// 创建了一个新的 3 行 3 列的Matrix实例
var mat = Matrix(rows: 3, columns: 3)

// 通过下标脚本设置值
mat[0,0] = 1.0
mat[0,1] = 2.0
mat[1,0] = 3.0
mat[1,1] = 5.0

// 通过下标脚本获取值
print("\(mat[0,0])")
print("\(mat[0,1])")
print("\(mat[1,0])")
print("\(mat[1,1])")
```

以上程序执行输出结果为：

```
1.0
2.0
3.0
5.0
```

Matrix 结构体提供了一个两个传入参数的构造方法，两个参数分别是rows和columns，创建了一个足够容纳rows * columns个数的Double类型数组。为了存储，将数组的大小和数组每个元素初始值0.0。

你可以通过传入合适的row和column的数量来构造一个新的Matrix实例。

Swift 继承

继承我们可以理解为一个类获取了另外一个类的方法和属性。

当一个类继承其它类时，继承类叫子类，被继承类叫超类（或父类）

在 Swift 中，类可以调用和访问超类的方法，属性和下标脚本，并且可以重写它们。

我们也可以为类中继承来的属性添加属性观察器。

基类

没有继承其它类的类，称之为基类（Base Class）。

以下实例中我们定义了基类 StudDetails ，描述了学生（stname）及其各科成绩的分数(mark1、mark2、mark3)：

```
class StudDetails {
    var stname: String!
    var mark1: Int!
    var mark2: Int!
    var mark3: Int!
    init(stname: String, mark1: Int, mark2: Int, mark3: Int) {
        self.stname = stname
        self.mark1 = mark1
        self.mark2 = mark2
        self.mark3 = mark3
    }
}
let stname = "swift"
let mark1 = 98
let mark2 = 89
let mark3 = 76

print(stname)
print(mark1)
print(mark2)
print(mark3)
```

以上程序执行输出结果为：

```
swift
98
89
76
```

```
swift
98
89
76
```

子类

子类指的是在一个已有类的基础上创建一个新的类。

为了指明某个类的超类，将超类名写在子类名的后面，用冒号(:)分隔,语法格式如下

```
class SomeClass: SomeSuperclass {
    // 类的定义
}
```

实例

以下实例中我们定义了超类 StudDetails，然后使用子类 Tom 继承它：

```
class StudDetails
{
    var mark1: Int;
    var mark2: Int;

    init(stm1:Int, results stm2:Int)
    {
        mark1 = stm1;
        mark2 = stm2;
    }

    func show()
    {
        print("Mark1:\(self.mark1), Mark2:\(self.mark2)")
    }
}

class Tom : StudDetails
{
    init()
    {
        super.init(stm1: 93, results: 89)
    }
}

let tom = Tom()
tom.show()
```

以上程序执行输出结果为：

```
Mark1:93,  Mark2:89
```

重写（Overriding）

子类可以通过继承来的实例方法，类方法，实例属性，或下标脚本来实现自己的定制功能，我们把这种行为叫重写（overriding）。

我们可以使用 override 关键字来实现重写。

访问超类的方法、属性及下标脚本

你可以通过使用super前缀来访问超类的方法，属性或下标脚本。

重写	访问方法，属性，下标脚本
方法	super.somemethod()
属性	super.someProperty()
下标脚本	super[someIndex]

重写方法和属性

重写方法

在我们的子类中我们可以使用 override 关键字来重写超类的方法。

以下实例中我们重写了 show() 方法：

```
class SuperClass {
    func show() {
        print("这是超类 SuperClass")
    }
}

class SubClass: SuperClass  {
    override func show() {
        print("这是子类 SubClass")
    }
}

let superClass = SuperClass()
superClass.show()

let subClass = SubClass()
subClass.show()
```

以上程序执行输出结果为：

```
这是超类  SuperClass
这是子类  SubClass
```

重写属性

你可以提供定制的 getter（或 setter）来重写任意继承来的属性，无论继承来的属性是存储型的还是计算型的属性。

子类并不知道继承来的属性是存储型的还是计算型的，它只知道继承来的属性会有一个名字和类型。所以你在重写一个属性时，必需将它的名字和类型都写出来。

注意点：

- 如果你在重写属性中提供了 setter，那么你也一定要提供 getter。
- 如果你不想在重写版本中的 getter 里修改继承来的属性值，你可以直接通过super.someProperty来返回继承来的值，其中someProperty是你要重写的属性的名字。

以下实例我们定义了超类 Circle 及子类 Rectangle, 在 Rectangle 类中我们重写属性 area：

```
class Circle {
    var radius = 12.5
    var area: String {
        return "矩形半径 \(radius) "
    }
}

// 继承超类 Circle
class Rectangle: Circle {
    var print = 7
    override var area: String {
        return super.area + " ，但现在被重写为 \(print)"
    }
}

let rect = Rectangle()
rect.radius = 25.0
rect.print = 3
print("Radius \(rect.area)")
```

以上程序执行输出结果为：

```
Radius 矩形半径 25.0 ，但现在被重写为 3
```

重写属性观察器

你可以在属性重写中为一个继承来的属性添加属性观察器。这样一来，当继承来的属性值发生改变时，你就会监测到。

注意：你不可以为继承来的常量存储型属性或继承来的只读计算型属性添加属性观察器。

```
class Circle {
    var radius = 12.5
    var area: String {
        return "矩形半径为 \(radius) "
    }
}

class Rectangle: Circle {
    var print = 7
    override var area: String {
        return super.area + " ，但现在被重写为 \(print)"
    }
}


let rect = Rectangle()
rect.radius = 25.0
rect.print = 3
print("半径: \(rect.area)")

class Square: Rectangle {
    override var radius: Double {
        didSet {
            print = Int(radius/5.0)+1
        }
    }
}


let sq = Square()
sq.radius = 100.0
print("半径: \(sq.area)")
```

```
半径:  矩形半径为  25.0  ，但现在被重写为  3
半径:  矩形半径为  100.0  ，但现在被重写为  21
```

防止重写

我们可以使用 final 关键字防止它们被重写。

如果你重写了final方法，属性或下标脚本，在编译时会报错。

你可以通过在关键字class前添加final特性（final class）来将整个类标记为 final 的，这样的类是不可被继承的，否则会报编译错误。

```
final class Circle {
    final var radius = 12.5
    var area: String {
        return "矩形半径为 \(radius) "
    }
}
class Rectangle: Circle {
    var print = 7
    override var area: String {
        return super.area + " ，但现在被重写为 \(print)"
    }
}

let rect = Rectangle()
rect.radius = 25.0
rect.print = 3
print("半径: \(rect.area)")

class Square: Rectangle {
    override var radius: Double {
        didSet {
            print = Int(radius/5.0)+1
        }
    }
}

let sq = Square()
sq.radius = 100.0
print("半径: \(sq.area)")
```

由于以上实例使用了 final 关键字不允许重写，所以执行会报错：

```
error: var overrides a 'final' var
    override var area: String {
                 ^
note: overridden declaration is here
    var area: String {
        ^
error: var overrides a 'final' var
    override var radius: Double {
                 ^
note: overridden declaration is here
    final var radius = 12.5
              ^
error: inheritance from a final class 'Circle'
class Rectangle: Circle {
      ^
```

Swift 构造过程

构造过程是为了使用某个类、结构体或枚举类型的实例而进行的准备过程。这个过程包含了为实例中的每个属性设置初始值和为其执行必要的准备和初始化任务。

Swift 构造函数使用 init() 方法。

与 Objective-C 中的构造器不同，Swift 的构造器无需返回值，它们的主要任务是保证新实例在第一次使用前完成正确的初始化。

类实例也可以通过定义析构器（deinitializer）在类实例释放之前执行清理内存的工作。

存储型属性的初始赋值

类和结构体在实例创建时，必须为所有存储型属性设置合适的初始值。

存储属性在构造器中赋值时，它们的值是被直接设置的，不会触发任何属性观测器。

存储属性在构造器中赋值流程：

- 创建初始值。
- 在属性定义中指定默认属性值。
- 初始化实例，并调用 init() 方法。

构造器

构造器在创建某特定类型的新实例时调用。它的最简形式类似于一个不带任何参数的实例方法，以关键字init命名。

语法

```
init()
{
    // 实例化后执行的代码
}
```

实例

以下结构体定义了一个不带参数的构造器 init，并在里面将存储型属性 length 和 breadth 的值初始化为 6 和 12：

```
struct rectangle {
    var length: Double
    var breadth: Double
    init() {
        length = 6
        breadth = 12
    }
}
var area = rectangle()
print("矩形面积为 \(area.length*area.breadth)")
```

以上程序执行输出结果为：

```
矩形面积为 72.0
```

默认属性值

我们可以在构造器中为存储型属性设置初始值；同样，也可以在属性声明时为其设置默认值。

> 使用默认值能让你的构造器更简洁、更清晰，且能通过默认值自动推导出属性的类型。

以下实例我们在属性声明时为其设置默认值：

```
struct rectangle {
    // 设置默认值
    var length = 6
    var breadth = 12
}
var area = rectangle()
print("矩形的面积为 \(area.length*area.breadth)")
```

以上程序执行输出结果为：

```
矩形面积为 72
```

构造参数

你可以在定义构造器 init() 时提供构造参数，如下所示：

```
struct Rectangle {
    var length: Double
    var breadth: Double
    var area: Double

    init(fromLength length: Double, fromBreadth breadth: Double) {
        self.length = length
        self.breadth = breadth
        area = length * breadth
    }

    init(fromLeng leng: Double, fromBread bread: Double) {
        self.length = leng
        self.breadth = bread
        area = leng * bread
    }
}

let ar = Rectangle(fromLength: 6, fromBreadth: 12)
print("面积为: \(ar.area)")

let are = Rectangle(fromLeng: 36, fromBread: 12)
print("面积为: \(are.area)")
```

以上程序执行输出结果为：

```
面积为:  72.0
面积为:  432.0
```

内部和外部参数名

跟函数和方法参数相同，构造参数也存在一个在构造器内部使用的参数名字和一个在调用构造器时使用的外部参数名字。

然而，构造器并不像函数和方法那样在括号前有一个可辨别的名字。所以在调用构造器时，主要通过构造器中的参数名和类型来确定需要调用的构造器。

如果你在定义构造器时没有提供参数的外部名字，Swift 会为每个构造器的参数自动生成一个跟内部名字相同的外部名。

```
struct Color {
    let red, green, blue: Double
    init(red: Double, green: Double, blue: Double) {
        self.red   = red
        self.green = green
        self.blue  = blue
    }
    init(white: Double) {
        red   = white
        green = white
        blue  = white
    }
}

// 创建一个新的Color实例，通过三种颜色的外部参数名来传值，并调用构造器
let magenta = Color(red: 1.0, green: 0.0, blue: 1.0)

print("red 值为: \(magenta.red)")
print("green 值为: \(magenta.green)")
print("blue 值为: \(magenta.blue)")

// 创建一个新的Color实例，通过三种颜色的外部参数名来传值，并调用构造器
let halfGray = Color(white: 0.5)
print("red 值为: \(halfGray.red)")
print("green 值为: \(halfGray.green)")
print("blue 值为: \(halfGray.blue)")
```

以上程序执行输出结果为：

```
red 值为: 1.0
green 值为: 0.0
blue 值为: 1.0
red 值为: 0.5
green 值为: 0.5
blue 值为: 0.5
```

没有外部名称参数

如果你不希望为构造器的某个参数提供外部名字，你可以使用下划线 `_` 来显示描述它的外部名。

```
struct Rectangle {
    var length: Double

    init(frombreadth breadth: Double) {
        length = breadth * 10
    }

    init(frombre bre: Double) {
        length = bre * 30
    }
    //不提供外部名字
    init(_ area: Double) {
        length = area
    }
}

// 调用不提供外部名字
let rectarea = Rectangle(180.0)
print("面积为: \(rectarea.length)")

// 调用不提供外部名字
let rearea = Rectangle(370.0)
print("面积为: \(rearea.length)")

// 调用不提供外部名字
let recarea = Rectangle(110.0)
print("面积为: \(recarea.length)")
```

以上程序执行输出结果为：

```
面积为: 180.0
面积为: 370.0
面积为: 110.0
```

可选属性类型

如果你定制的类型包含一个逻辑上允许取值为空的存储型属性，你都需要将它定义为可选类型optional type（可选属性类型）。

当存储属性声明为可选时，将自动初始化为空 nil。

```
struct Rectangle {
    var length: Double?

    init(frombreadth breadth: Double) {
        length = breadth * 10
    }

    init(frombre bre: Double) {
        length = bre * 30
    }

    init(_ area: Double) {
        length = area
    }
}

let rectarea = Rectangle(180.0)
print("面积为：\(rectarea.length)")

let rearea = Rectangle(370.0)
print("面积为：\(rearea.length)")

let recarea = Rectangle(110.0)
print("面积为：\(recarea.length)")
```

以上程序执行输出结果为：

```
面积为：Optional(180.0)
面积为：Optional(370.0)
面积为：Optional(110.0)
```

构造过程中修改常量属性

只要在构造过程结束前常量的值能确定，你可以在构造过程中的任意时间点修改常量属性的值。

对某个类实例来说，它的常量属性只能在定义它的类的构造过程中修改；不能在子类中修改。

尽管 length 属性现在是常量，我们仍然可以在其类的构造器中设置它的值：

```
struct Rectangle {
    let length: Double?

    init(frombreadth breadth: Double) {
        length = breadth * 10
    }

    init(frombre bre: Double) {
        length = bre * 30
    }

    init(_ area: Double) {
        length = area
    }
}

let rectarea = Rectangle(180.0)
print("面积为：\(rectarea.length)")

let rearea = Rectangle(370.0)
print("面积为：\(rearea.length)")

let recarea = Rectangle(110.0)
print("面积为：\(recarea.length)")
```

以上程序执行输出结果为：

```
面积为：Optional(180.0)
面积为：Optional(370.0)
面积为：Optional(110.0)
```

默认构造器

默认构造器将简单的创建一个所有属性值都设置为默认值的实例:

以下实例中，ShoppingListItem类中的所有属性都有默认值，且它是没有父类的基类，它将自动获得一个可以为所有属性设置默认值的默认构造器

```
class ShoppingListItem {
    var name: String?
    var quantity = 1
    var purchased = false
}
var item = ShoppingListItem()


print("名字为: \(item.name)")
print("数理为: \(item.quantity)")
print("是否付款: \(item.purchased)")
```

以上程序执行输出结果为：

```
名字为:  nil
数理为:  1
是否付款:  false
```

结构体的逐一成员构造器

如果结构体对所有存储型属性提供了默认值且自身没有提供定制的构造器，它们能自动获得一个逐一成员构造器。

我们在调用逐一成员构造器时，通过与成员属性名相同的参数名进行传值来完成对成员属性的初始赋值。

下面例子中定义了一个结构体 Rectangle，它包含两个属性 length 和 breadth。Swift 可以根据这两个属性的初始赋值100.0 、200.0自动推导出它们的类型Double。

```
struct Rectangle {
    var length = 100.0, breadth = 200.0
}
let area = Rectangle(length: 24.0, breadth: 32.0)

print("矩形的面积: \(area.length)")
print("矩形的面积: \(area.breadth)")
```

由于这两个存储型属性都有默认值，结构体 Rectangle 自动获得了一个逐一成员构造器 init(width:height:)。 你可以用它来为 Rectangle 创建新的实例。

以上程序执行输出结果为：

```
名字为:  nil
矩形的面积:  24.0
矩形的面积:  32.0
```

值类型的构造器代理

构造器可以通过调用其它构造器来完成实例的部分构造过程。这一过程称为构造器代理，它能减少多个构造器间的代码重复。

以下实例中，Rect 结构体调用了 Size 和 Point 的构造过程：

```
struct Size {
    var width = 0.0, height = 0.0
}
struct Point {
    var x = 0.0, y = 0.0
}

struct Rect {
    var origin = Point()
    var size = Size()
    init() {}
    init(origin: Point, size: Size) {
        self.origin = origin
        self.size = size
    }
    init(center: Point, size: Size) {
        let originX = center.x - (size.width / 2)
        let originY = center.y - (size.height / 2)
        self.init(origin: Point(x: originX, y: originY), size: size
    }
}


// origin和size属性都使用定义时的默认值Point(x: 0.0, y: 0.0)和Size(widt
let basicRect = Rect()
print("Size 结构体初始值: \(basicRect.size.width, basicRect.size.heig
print("Rect 结构体初始值: \(basicRect.origin.x, basicRect.origin.y) '

// 将origin和size的参数值赋给对应的存储型属性
let originRect = Rect(origin: Point(x: 2.0, y: 2.0),
    size: Size(width: 5.0, height: 5.0))

print("Size 结构体初始值: \(originRect.size.width, originRect.size.he
print("Rect 结构体初始值: \(originRect.origin.x, originRect.origin.y)


//先通过center和size的值计算出origin的坐标。
//然后再调用（或代理给）init(origin:size:)构造器来将新的origin和size值赋值
let centerRect = Rect(center: Point(x: 4.0, y: 4.0),
    size: Size(width: 3.0, height: 3.0))

print("Size 结构体初始值: \(centerRect.size.width, centerRect.size.he
print("Rect 结构体初始值: \(centerRect.origin.x, centerRect.origin.y)
```

以上程序执行输出结果为：

```
Size 结构体初始值: (0.0, 0.0)
Rect 结构体初始值: (0.0, 0.0)
Size 结构体初始值: (5.0, 5.0)
Rect 结构体初始值: (2.0, 2.0)
Size 结构体初始值: (3.0, 3.0)
Rect 结构体初始值: (2.5, 2.5)
```

构造器代理规则

值类型	类类型
不支持继承，所以构造器代理的过程相对简单，因为它们只能代理给本身提供的其它构造器。 你可以使用self.init在自定义的构造器中引用其它的属于相同值类型的构造器。	它可以继承自其它类,这意味着类有责任保证其所有继承的存储型属性在构造时也能正确的初始化。

类的继承和构造过程

Swift 提供了两种类型的类构造器来确保所有类实例中存储型属性都能获得初始值，它们分别是指定构造器和便利构造器。

指定构造器	便利构造器
类中最主要的构造器	类中比较次要的、辅助型的构造器
初始化类中提供的所有属性，并根据父类链往上调用父类的构造器来实现父类的初始化。	可以定义便利构造器来调用同一个类中 并为其参数提供默认值。你也可以定义 建一个特殊用途或特定输入的实例。
每一个类都必须拥有至少一个指定构造器	只在必要的时候为类提供便利构造器
`Init(parameters) { statements }`	`convenience init(parameters) {`

指定构造器实例

```
class mainClass {
    var no1 : Int // 局部存储变量
    init(no1 : Int) {
        self.no1 = no1 // 初始化
    }
}
class subClass : mainClass {
    var no2 : Int // 新的子类存储变量
    init(no1 : Int, no2 : Int) {
        self.no2 = no2 // 初始化
        super.init(no1:no1) // 初始化超类
    }
}

let res = mainClass(no1: 10)
let res2 = subClass(no1: 10, no2: 20)

print("res 为: \(res.no1)")
print("res2 为: \(res2.no1)")
print("res2 为: \(res2.no2)")
```

以上程序执行输出结果为：

```
res 为: 10
res 为: 10
res 为: 20
```

便利构造器实例

```
class mainClass {
    var no1 : Int // 局部存储变量
    init(no1 : Int) {
        self.no1 = no1 // 初始化
    }
}

class subClass : mainClass {
    var no2 : Int
    init(no1 : Int, no2 : Int) {
        self.no2 = no2
        super.init(no1:no1)
    }
    // 便利方法只需要一个参数
    override convenience init(no1: Int)  {
        self.init(no1:no1, no2:0)
    }
}
let res = mainClass(no1: 20)
let res2 = subClass(no1: 30, no2: 50)

print("res 为: \(res.no1)")
print("res2 为: \(res2.no1)")
print("res2 为: \(res2.no2)")
```

以上程序执行输出结果为：

```
res 为:  20
res2 为:  30
res2 为:  50
```

构造器的继承和重载

Swift 中的子类不会默认继承父类的构造器。

父类的构造器仅在确定和安全的情况下被继承。

当你重写一个父类指定构造器时，你需要写override修饰符。

```
class SuperClass {
    var corners = 4
    var description: String {
        return "\(corners) 边"
    }
}
let rectangle = SuperClass()
print("矩形: \(rectangle.description)")

class SubClass: SuperClass {
    override init() {  //重载构造器
        super.init()
        corners = 5
    }
}

let subClass = SubClass()
print("五角型: \(subClass.description)")
```

以上程序执行输出结果为：

```
矩形:  4  边
五角型:  5  边
```

指定构造器和便利构造器实例

接下来的例子将在操作中展示指定构造器、便利构造器和自动构造器的继承。

它定义了包含两个个类MainClass、SubClass的类层次结构，并将演示它们的构造器是如何相互作用的。

```
class MainClass {
    var name: String

    init(name: String) {
        self.name = name
    }

    convenience init() {
        self.init(name: "[匿名]")
    }
}
let main = MainClass(name: "Runoob")
print("MainClass 名字为: \(main.name)")

let main2 = MainClass()
print("没有对应名字: \(main2.name)")

class SubClass: MainClass {
    var count: Int
    init(name: String, count: Int) {
        self.count = count
        super.init(name: name)
    }

    override convenience init(name: String) {
        self.init(name: name, count: 1)
    }
}

let sub = SubClass(name: "Runoob")
print("MainClass 名字为: \(sub.name)")

let sub2 = SubClass(name: "Runoob", count: 3)
print("count 变量: \(sub2.count)")
```

以上程序执行输出结果为：

```
MainClass 名字为: Runoob
没有对应名字: [匿名]
MainClass 名字为: Runoob
count 变量: 3
```

类的可失败构造器

如果一个类，结构体或枚举类型的对象，在构造自身的过程中有可能失败，则为其定义一个可失败构造器。

变量初始化失败可能的原因有：

- 传入无效的参数值。
- 缺少某种所需的外部资源。
- 没有满足特定条件。

为了妥善处理这种构造过程中可能会失败的情况。

你可以在一个类，结构体或是枚举类型的定义中，添加一个或多个可失败构造器。其语法为在init关键字后面加添问号(init?)。

实例

下例中，定义了一个名为Animal的结构体，其中有一个名为species的，String类型的常量属性。

同时该结构体还定义了一个，带一个String类型参数species的,可失败构造器。这个可失败构造器，被用来检查传入的参数是否为一个空字符串，如果为空字符串，则该可失败构造器，构建对象失败，否则成功。

```
struct Animal {
    let species: String
    init?(species: String) {
        if species.isEmpty { return nil }
        self.species = species
    }
}

//通过该可失败构造器来构建一个Animal的对象，并检查其构建过程是否成功
// someCreature 的类型是 Animal? 而不是 Animal
let someCreature = Animal(species: "长颈鹿")

// 打印 "动物初始化为长颈鹿"
if let giraffe = someCreature {
    print("动物初始化为\(giraffe.species)")
}
```

以上程序执行输出结果为：

```
动物初始化为长颈鹿
```

枚举类型的可失败构造器

你可以通过构造一个带一个或多个参数的可失败构造器来获取枚举类型中特定的枚举成员。

实例

下例中，定义了一个名为TemperatureUnit的枚举类型。其中包含了三个可能的枚举成员(Kelvin，Celsius，和 Fahrenheit)和一个被用来找到Character值所对应的枚举成员的可失败构造器：

```
enum TemperatureUnit {
    // 开尔文，摄氏，华氏
    case Kelvin, Celsius, Fahrenheit
    init?(symbol: Character) {
        switch symbol {
        case "K":
            self = .Kelvin
        case "C":
            self = .Celsius
        case "F":
            self = .Fahrenheit
        default:
            return nil
        }
    }
}


let fahrenheitUnit = TemperatureUnit(symbol: "F")
if fahrenheitUnit != nil {
    print("这是一个已定义的温度单位，所以初始化成功。")
}

let unknownUnit = TemperatureUnit(symbol: "X")
if unknownUnit == nil {
    print("这不是一个已定义的温度单位，所以初始化失败。")
}
```

以上程序执行输出结果为：

```
这是一个已定义的温度单位，所以初始化成功。
这不是一个已定义的温度单位，所以初始化失败。
```

类的可失败构造器

值类型（如结构体或枚举类型）的可失败构造器，对何时何地触发构造失败这个行为没有任何的限制。

但是，类的可失败构造器只能在所有的类属性被初始化后和所有类之间的构造器之间的代理调用发生完后触发失败行为。

实例

下例子中，定义了一个名为 StudRecord 的类，因为 studname 属性是一个常量，所以一旦 StudRecord 类构造成功，studname 属性肯定有一个非nil的值。

```
class StudRecord {
    let studname: String!
    init?(studname: String) {
        self.studname = studname
        if studname.isEmpty { return nil }
    }
}
if let stname = StudRecord(studname: "失败构造器") {
    print("模块为 \(stname.studname)")
}
```

以上程序执行输出结果为：

```
模块为  失败构造器
```

覆盖一个可失败构造器

就如同其它构造器一样，你也可以用子类的可失败构造器覆盖基类的可失败构造器。

者你也可以用子类的非可失败构造器覆盖一个基类的可失败构造器。

你可以用一个非可失败构造器覆盖一个可失败构造器，但反过来却行不通。

一个非可失败的构造器永远也不能代理调用一个可失败构造器。

实例

以下实例描述了可失败与非可失败构造器：

```
class Planet {
    var name: String

    init(name: String) {
        self.name = name
    }

    convenience init() {
        self.init(name: "[No Planets]")
    }
}
let plName = Planet(name: "Mercury")
print("行星的名字是: \(plName.name)")

let noplName = Planet()
print("没有这个名字的行星: \(noplName.name)")

class planets: Planet {
    var count: Int

    init(name: String, count: Int) {
        self.count = count
        super.init(name: name)
    }

    override convenience init(name: String) {
        self.init(name: name, count: 1)
    }
}
```

以上程序执行输出结果为：

```
行星的名字是: Mercury
没有这个名字的行星: [No Planets]
```

可失败构造器 init!

通常来说我们通过在init关键字后添加问号的方式（init?）来定义一个可失败构造器，但你也可以使用通过在init后面添加惊叹号的方式来定义一个可失败构造器(init!)。实例如下：

```
struct StudRecord {
    let stname: String

    init!(stname: String) {
        if stname.isEmpty {return nil }
        self.stname = stname
    }
}

let stmark = StudRecord(stname: "Runoob")
if let name = stmark {
    print("指定了学生名")
}

let blankname = StudRecord(stname: "")
if blankname == nil {
    print("学生名为空")
}
```

以上程序执行输出结果为：

```
指定了学生名  学生名为空
```

Swift 析构过程

在一个类的实例被释放之前，析构函数被立即调用。用关键字 `deinit` 来标示析构函数，类似于初始化函数用 `init` 来标示。析构函数只适用于类类型。

析构过程原理

Swift 会自动释放不再需要的实例以释放资源。

Swift 通过自动引用计数（ARC）处理实例的内存管理。

通常当你的实例被释放时不需要手动地去清理。但是，当使用自己的资源时，你可能需要进行一些额外的清理。

例如，如果创建了一个自定义的类来打开一个文件，并写入一些数据，你可能需要在类实例被释放之前关闭该文件。

语法

在类的定义中，每个类最多只能有一个析构函数。析构函数不带任何参数，在写法上不带括号：

```
deinit {
    // 执行析构过程
}
```

实例

```
var counter = 0;  // 引用计数器
class BaseClass {
    init() {
        counter++;
    }
    deinit {
        counter--;
    }
}

var show: BaseClass? = BaseClass()
print(counter)
show = nil
print(counter)
```

以上程序执行输出结果为：

```
1
0
```

当 show = nil 语句执行后，计算器减去 1，show 占用的内存就会释放。

```
var counter = 0;  // 引用计数器

class BaseClass {
    init() {
        counter++;
    }

    deinit {
        counter--;
    }
}

var show: BaseClass? = BaseClass()

print(counter)
print(counter)
```

以上程序执行输出结果为：

```
1
1
```

Swift 可选链

可选链（Optional Chaining）是一种是一种可以请求和调用属性、方法和子脚本的过程，用于请求或调用的目标可能为nil。

可选链返回两个值：

- 如果目标有值，调用就会成功，返回该值
- 如果目标为nil，调用将返回nil

多次请求或调用可以被链接成一个链，如果任意一个节点为nil将导致整条链失效。

可选链可替代强制解析

通过在属性、方法、或下标脚本的可选值后面放一个问号(?)，即可定义一个可选链。

可选链 **'?'**	感叹号（**!**）强制展开方法，属性，下标脚本可选链
? 放置于可选值后来调用方法，属性，下标脚本	! 放置于可选值后来调用方法，属性，下标脚本来强制展开值
当可选为 nil 输出比较友好的错误信息	当可选为 nil 时强制展开执行错误

使用感叹号(!)可选链实例

```
class Person {
    var residence: Residence?
}

class Residence {
    var numberOfRooms = 1
}

let john = Person()

//将导致运行时错误
let roomCount = john.residence!.numberOfRooms
```

以上程序执行输出结果为：

```
fatal error: unexpectedly found nil while unwrapping an Optional va
```

使用感叹号(!)可选链实例

```
class Person {
    var residence: Residence?
}

class Residence {
    var numberOfRooms = 1
}

let john = Person()

// 链接可选residence?属性，如果residence存在则取回numberOfRooms的值
if let roomCount = john.residence?.numberOfRooms {
    print("John 的房间号为 \(roomCount)。")
} else {
    print("不能查看房间号")
}
```

以上程序执行输出结果为：

```
不能查看房间号
```

因为这种尝试获得numberOfRooms的操作有可能失败，可选链会返回Int?类型值，或者称作"可选Int"。当residence是空的时候（上例），选择Int将会为空，因此会出现无法访问numberOfRooms的情况。

要注意的是，即使numberOfRooms是非可选Int（Int?）时这一点也成立。只要是通过可选链的请求就意味着最后numberOfRooms总是返回一个Int?而不是Int。

为可选链定义模型类

你可以使用可选链来多层调用属性，方法，和下标脚本。这让你可以利用它们之间的复杂模型来获取更底层的属性，并检查是否可以成功获取此类底层属性。

实例

定义了四个模型类，其中包括多层可选链：

```
class Person {
    var residence: Residence?
}

// 定义了一个变量 rooms，它被初始化为一个Room[]类型的空数组
class Residence {
    var rooms = [Room]()
    var numberOfRooms: Int {
        return rooms.count
    }
    subscript(i: Int) -> Room {
        return rooms[i]
    }
    func printNumberOfRooms() {
        print("房间号为 \(numberOfRooms)")
    }
    var address: Address?
}

// Room 定义一个name属性和一个设定room名的初始化器
class Room {
    let name: String
    init(name: String) { self.name = name }
}

// 模型中的最终类叫做Address
class Address {
    var buildingName: String?
    var buildingNumber: String?
    var street: String?
    func buildingIdentifier() -> String? {
        if (buildingName != nil) {
            return buildingName
        } else if (buildingNumber != nil) {
            return buildingNumber
        } else {
            return nil
        }
    }
}
```

通过可选链调用方法

你可以使用可选链的来调用可选值的方法并检查方法调用是否成功。即使这个方法没有返回值，你依然可以使用可选链来达成这一目的。

```
class Person {
    var residence: Residence?
}

// 定义了一个变量 rooms，它被初始化为一个Room[]类型的空数组
class Residence {
    var rooms = [Room]()
    var numberOfRooms: Int {
        return rooms.count
    }
    subscript(i: Int) -> Room {
        return rooms[i]
    }
    func printNumberOfRooms() {
        print("房间号为 \(numberOfRooms)")
    }
    var address: Address?
}

// Room 定义一个name属性和一个设定room名的初始化器
class Room {
    let name: String
    init(name: String) { self.name = name }
}

// 模型中的最终类叫做Address
class Address {
    var buildingName: String?
    var buildingNumber: String?
    var street: String?
    func buildingIdentifier() -> String? {
        if (buildingName != nil) {
            return buildingName
        } else if (buildingNumber != nil) {
            return buildingNumber
        } else {
            return nil
        }
    }
}

let john = Person()

if ((john.residence?.printNumberOfRooms()) != nil) {
    print("输出房间号")
} else {
    print("无法输出房间号")
}
```

以上程序执行输出结果为：

```
无法输出房间号
```

使用if语句来检查是否能成功调用printNumberOfRooms方法：如果方法通过可选链调用成功，printNumberOfRooms的隐式返回值将会是Void，如果没有成功，将返回nil。

使用可选链调用下标脚本

你可以使用可选链来尝试从下标脚本获取值并检查下标脚本的调用是否成功，然而，你不能通过可选链来设置下标脚本。

实例1

```
class Person {
    var residence: Residence?
}

// 定义了一个变量 rooms，它被初始化为一个Room[]类型的空数组
class Residence {
    var rooms = [Room]()
    var numberOfRooms: Int {
        return rooms.count
    }
    subscript(i: Int) -> Room {
        return rooms[i]
    }
    func printNumberOfRooms() {
        print("房间号为 \(numberOfRooms)")
    }
    var address: Address?
}

// Room 定义一个name属性和一个设定room名的初始化器
class Room {
    let name: String
    init(name: String) { self.name = name }
}

// 模型中的最终类叫做Address
class Address {
    var buildingName: String?
    var buildingNumber: String?
    var street: String?
    func buildingIdentifier() -> String? {
        if (buildingName != nil) {
            return buildingName
        } else if (buildingNumber != nil) {
            return buildingNumber
        } else {
            return nil
        }
    }
}

let john = Person()
if let firstRoomName = john.residence?[0].name {
    print("第一个房间名 \(firstRoomName).")
} else {
    print("无法检索到房间")
}
```

以上程序执行输出结果为：

```
无法检索到房间
```

在下标脚本调用中可选链的问号直接跟在 circname.print 的后面，在下标脚本括号的前面，因为circname.print是可选链试图获得的可选值。

实例2

实例中创建一个Residence实例给john.residence，且在他的rooms数组中有一个或多个Room实例，那么你可以使用可选链通过Residence下标脚本来获取在rooms数组中的实例了：

```
class Person {
    var residence: Residence?
}

// 定义了一个变量 rooms，它被初始化为一个Room[]类型的空数组
class Residence {
    var rooms = [Room]()
    var numberOfRooms: Int {
        return rooms.count
    }
    subscript(i: Int) -> Room {
        return rooms[i]
    }
    func printNumberOfRooms() {
        print("房间号为 \(numberOfRooms)")
    }
    var address: Address?
}

// Room 定义一个name属性和一个设定room名的初始化器
class Room {
    let name: String
    init(name: String) { self.name = name }
}

// 模型中的最终类叫做Address
class Address {
    var buildingName: String?
    var buildingNumber: String?
    var street: String?
    func buildingIdentifier() -> String? {
        if (buildingName != nil) {
            return buildingName
        } else if (buildingNumber != nil) {
            return buildingNumber
        } else {
            return nil
        }
```

```
    }
}

let john = Person()
let johnsHouse = Residence()
johnsHouse.rooms.append(Room(name: "客厅"))
johnsHouse.rooms.append(Room(name: "厨房"))
john.residence = johnsHouse

if let firstRoomName = john.residence?[0].name {
    print("第一个房间名为\(firstRoomName)")
} else {
    print("无法检索到房间")
}
```

以上程序执行输出结果为：

```
第一个房间名为客厅
```

通过可选链接调用来访问下标

通过可选链接调用，我们可以用下标来对可选值进行读取或写入，并且判断下标调用是否成功。

实例

```
class Person {
    var residence: Residence?
}

// 定义了一个变量 rooms，它被初始化为一个Room[]类型的空数组
class Residence {
    var rooms = [Room]()
    var numberOfRooms: Int {
        return rooms.count
    }
    subscript(i: Int) -> Room {
        return rooms[i]
    }
    func printNumberOfRooms() {
        print("房间号为 \(numberOfRooms)")
    }
    var address: Address?
}

// Room 定义一个name属性和一个设定room名的初始化器
class Room {
```

```
    let name: String
    init(name: String) { self.name = name }
}

// 模型中的最终类叫做Address
class Address {
    var buildingName: String?
    var buildingNumber: String?
    var street: String?
    func buildingIdentifier() -> String? {
        if (buildingName != nil) {
            return buildingName
        } else if (buildingNumber != nil) {
            return buildingNumber
        } else {
            return nil
        }
    }
}

let john = Person()

let johnsHouse = Residence()
johnsHouse.rooms.append(Room(name: "客厅"))
johnsHouse.rooms.append(Room(name: "厨房"))
john.residence = johnsHouse

if let firstRoomName = john.residence?[0].name {
    print("第一个房间名为\(firstRoomName)")
} else {
    print("无法检索到房间")
}
```

以上程序执行输出结果为：

```
第一个房间名为客厅
```

访问可选类型的下标

如果下标返回可空类型值，比如Swift中Dictionary的key下标。可以在下标的闭合括号后面放一个问号来链接下标的可空返回值：

```
var testScores = ["Dave": [86, 82, 84], "Bev": [79, 94, 81]]
testScores["Dave"]?[0] = 91
testScores["Bev"]?[0]++
testScores["Brian"]?[0] = 72
// the "Dave" array is now [91, 82, 84] and the "Bev" array is now
```

上面的例子中定义了一个testScores数组，包含了两个键值对， 把String类型的key映射到一个整形数组。

这个例子用可选链接调用把"Dave"数组中第一个元素设为91，把"Bev"数组的第一个元素+1，然后尝试把"Brian"数组中的第一个元素设为72。

前两个调用是成功的，因为这两个key存在。但是key"Brian"在字典中不存在，所以第三个调用失败。

连接多层链接

你可以将多层可选链连接在一起，可以掘取模型内更下层的属性方法和下标脚本。然而多层可选链不能再添加比已经返回的可选值更多的层。

如果你试图通过可选链获得Int值，不论使用了多少层链接返回的总是Int?。 相似的，如果你试图通过可选链获得Int?值，不论使用了多少层链接返回的总是Int?。

实例1

下面的例子试图获取john的residence属性里的address的street属性。这里使用了两层可选链来联系residence和address属性，它们两者都是可选类型：

```
class Person {
    var residence: Residence?
}

// 定义了一个变量 rooms，它被初始化为一个Room[]类型的空数组
class Residence {
    var rooms = [Room]()
    var numberOfRooms: Int {
        return rooms.count
    }
    subscript(i: Int) -> Room {
        return rooms[i]
    }
    func printNumberOfRooms() {
        print("房间号为 \(numberOfRooms)")
    }
    var address: Address?
}

// Room 定义一个name属性和一个设定room名的初始化器
class Room {
    let name: String
    init(name: String) { self.name = name }
}

// 模型中的最终类叫做Address
class Address {
    var buildingName: String?
    var buildingNumber: String?
    var street: String?
    func buildingIdentifier() -> String? {
        if (buildingName != nil) {
            return buildingName
        } else if (buildingNumber != nil) {
            return buildingNumber
        } else {
            return nil
        }
    }
}

let john = Person()

if let johnsStreet = john.residence?.address?.street {
    print("John 的地址为 \(johnsStreet).")
} else {
    print("不能检索地址")
}
```

以上程序执行输出结果为：

```
不能检索地址
```

实例2

如果你为Address设定一个实例来作为john.residence.address的值，并为address的street属性设定一个实际值，你可以通过多层可选链来得到这个属性值。

```
class Person {
   var residence: Residence?
}

class Residence {

    var rooms = [Room]()
    var numberOfRooms: Int {
        return rooms.count
    }
    subscript(i: Int) -> Room {
        get{
            return rooms[i]
        }
        set {
            rooms[i] = newValue
        }
    }
    func printNumberOfRooms() {
        print("房间号为 \(numberOfRooms)")
    }
    var address: Address?
}

class Room {
    let name: String
    init(name: String) { self.name = name }
}

class Address {
    var buildingName: String?
    var buildingNumber: String?
    var street: String?
    func buildingIdentifier() -> String? {
        if (buildingName != nil) {
            return buildingName
        } else if (buildingNumber != nil) {
            return buildingNumber
        } else {
            return nil
        }
    }
}
```

```
let john = Person()
john.residence?[0] = Room(name: "浴室")

let johnsHouse = Residence()
johnsHouse.rooms.append(Room(name: "客厅"))
johnsHouse.rooms.append(Room(name: "厨房"))
john.residence = johnsHouse

if let firstRoomName = john.residence?[0].name {
    print("第一个房间是\(firstRoomName)")
} else {
    print("无法检索房间")
}
```

以上实例输出结果为：

```
第一个房间是客厅
```

对返回可选值的函数进行链接

我们还可以通过可选链接来调用返回可空值的方法，并且可以继续对可选值进行链接。

实例

```
class Person {
    var residence: Residence?
}

// 定义了一个变量 rooms，它被初始化为一个Room[]类型的空数组
class Residence {
    var rooms = [Room]()
    var numberOfRooms: Int {
        return rooms.count
    }
    subscript(i: Int) -> Room {
        return rooms[i]
    }
    func printNumberOfRooms() {
        print("房间号为 \(numberOfRooms)")
    }
    var address: Address?
}

// Room 定义一个name属性和一个设定room名的初始化器
class Room {
    let name: String
    init(name: String) { self.name = name }
}

// 模型中的最终类叫做Address
class Address {
    var buildingName: String?
    var buildingNumber: String?
    var street: String?
    func buildingIdentifier() -> String? {
        if (buildingName != nil) {
            return buildingName
        } else if (buildingNumber != nil) {
            return buildingNumber
        } else {
            return nil
        }
    }
}

let john = Person()

if john.residence?.printNumberOfRooms() != nil {
    print("指定了房间号)")
}  else {
    print("未指定房间号")
}
```

以上程序执行输出结果为：

```
未指定房间号
```

Swift 自动引用计数（ARC）

Swift 使用自动引用计数（ARC）这一机制来跟踪和管理应用程序的内存

通常情况下我们不需要去手动释放内存，因为 ARC 会在类的实例不再被使用时，自动释放其占用的内存。

但在有些时候我们还是需要在代码中实现内存管理。

ARC 功能

- 当每次使用 init() 方法创建一个类的新的实例的时候，ARC 会分配一大块内存用来储存实例的信息。
- 内存中会包含实例的类型信息，以及这个实例所有相关属性的值。
- 当实例不再被使用时，ARC 释放实例所占用的内存，并让释放的内存能挪作他用。
- 为了确保使用中的实例不会被销毁，ARC 会跟踪和计算每一个实例正在被多少属性，常量和变量所引用。
- 实例赋值给属性、常量或变量，它们都会创建此实例的强引用，只要强引用还在，实例是不允许被销毁的。

ARC 实例

```
class Person {
    let name: String
    init(name: String) {
        self.name = name
        print("\(name) 开始初始化")
    }
    deinit {
        print("\(name) 被析构")
    }
}

// 值会被自动初始化为nil，目前还不会引用到Person类的实例
var reference1: Person?
var reference2: Person?
var reference3: Person?

// 创建Person类的新实例
reference1 = Person(name: "Runoob")

//赋值给其他两个变量，该实例又会多出两个强引用
reference2 = reference1
reference3 = reference1

//断开第一个强引用
reference1 = nil
//断开第二个强引用
reference2 = nil
//断开第三个强引用，并调用析构函数
reference3 = nil
```

以上程序执行输出结果为：

```
Runoob 开始初始化
Runoob 被析构
```

类实例之间的循环强引用

在上面的例子中，ARC 会跟踪你所新创建的 Person 实例的引用数量，并且会在 Person 实例不再被需要时销毁它。

然而，我们可能会写出这样的代码，一个类永远不会有0个强引用。这种情况发生在两个类实例互相保持对方的强引用，并让对方不被销毁。这就是所谓的循环强引用。

实例

下面展示了一个不经意产生循环强引用的例子。例子定义了两个类：Person和Apartment，用来建模公寓和它其中的居民:

```
class Person {
    let name: String
    init(name: String) { self.name = name }
    var apartment: Apartment?
    deinit { print("\(name) 被析构") }
}

class Apartment {
    let number: Int
    init(number: Int) { self.number = number }
    var tenant: Person?
    deinit { print("Apartment #\(number) 被析构") }
}

// 两个变量都被初始化为nil
var runoob: Person?
var number73: Apartment?

// 赋值
runoob = Person(name: "Runoob")
number73 = Apartment(number: 73)

// 意感叹号是用来展开和访问可选变量 runoob 和 number73 中的实例
// 循环强引用被创建
runoob!.apartment = number73
number73!.tenant = runoob

// 断开 runoob 和 number73 变量所持有的强引用时，引用计数并不会降为 0，实例
// 注意，当你把这两个变量设为nil时，没有任何一个析构函数被调用。
// 强引用循环阻止了Person和Apartment类实例的销毁，并在你的应用程序中造成了内
runoob = nil
number73 = nil
```

解决实例之间的循环强引用

Swift 提供了两种办法用来解决你在使用类的属性时所遇到的循环强引用问题：

- 弱引用
- 无主引用

弱引用和无主引用允许循环引用中的一个实例引用另外一个实例而不保持强引用。这样实例能够互相引用而不产生循环强引用。

对于生命周期中会变为nil的实例使用弱引用。相反的，对于初始化赋值后再也不会被赋值为nil的实例，使用无主引用。

弱引用实例

```
class Module {
    let name: String
    init(name: String) { self.name = name }
    var sub: SubModule?
    deinit { print("\(name) 主模块") }
}

class SubModule {
    let number: Int

    init(number: Int) { self.number = number }

    weak var topic: Module?

    deinit { print("子模块 topic 数为 \(number)") }
}

var toc: Module?
var list: SubModule?
toc = Module(name: "ARC")
list = SubModule(number: 4)
toc!.sub = list
list!.topic = toc

toc = nil
list = nil
```

以上程序执行输出结果为：

```
ARC 主模块
子模块 topic 数为 4
```

无主引用实例

```
class Student {
    let name: String
    var section: Marks?

    init(name: String) {
        self.name = name
    }

    deinit { print("\(name)") }
}
class Marks {
    let marks: Int
    unowned let stname: Student

    init(marks: Int, stname: Student) {
        self.marks = marks
        self.stname = stname
    }

    deinit { print("学生的分数为 \(marks)") }
}

var module: Student?
module = Student(name: "ARC")
module!.section = Marks(marks: 98, stname: module!)
module = nil
```

以上程序执行输出结果为：

```
ARC
学生的分数为 98
```

闭包引起的循环强引用

循环强引用还会发生在当你将一个闭包赋值给类实例的某个属性，并且这个闭包体中又使用了实例。这个闭包体中可能访问了实例的某个属性，例如self.someProperty，或者闭包中调用了实例的某个方法，例如self.someMethod。这两种情况都导致了闭包 "捕获" self，从而产生了循环强引用。

实例

下面的例子为你展示了当一个闭包引用了self后是如何产生一个循环强引用的。例子中定义了一个叫HTMLElement的类，用一种简单的模型表示 HTML 中的一个单独的元素：

```
class HTMLElement {

    let name: String
    let text: String?

    lazy var asHTML: () -> String = {
        if let text = self.text {
            return "<\(self.name)>\(text)</\(self.name)>"
        } else {
            return "<\(self.name) />"
        }
    }

    init(name: String, text: String? = nil) {
        self.name = name
        self.text = text
    }

    deinit {
        print("\(name) is being deinitialized")
    }

}

// 创建实例并打印信息
var paragraph: HTMLElement? = HTMLElement(name: "p", text: "hello,
print(paragraph!.asHTML())
```

HTMLElement 类产生了类实例和 asHTML 默认值的闭包之间的循环强引用。

实例的 asHTML 属性持有闭包的强引用。但是，闭包在其闭包体内使用了self（引用了self.name和self.text），因此闭包捕获了self，这意味着闭包又反过来持有了HTMLElement实例的强引用。这样两个对象就产生了循环强引用。

解决闭包引起的循环强引用:在定义闭包时同时定义捕获列表作为闭包的一部分，通过这种方式可以解决闭包和类实例之间的循环强引用。

弱引用和无主引用

当闭包和捕获的实例总是互相引用时并且总是同时销毁时，将闭包内的捕获定义为无主引用。

相反的，当捕获引用有时可能会是nil时，将闭包内的捕获定义为弱引用。

如果捕获的引用绝对不会置为nil，应该用无主引用，而不是弱引用。

实例

前面的HTMLElement例子中，无主引用是正确的解决循环强引用的方法。这样编写HTMLElement类来避免循环强引用：

```
class HTMLElement {

    let name: String
    let text: String?

    lazy var asHTML: () -> String = {
        [unowned self] in
        if let text = self.text {
            return "<\(self.name)>\(text)</\(self.name)>"
        } else {
            return "<\(self.name) />"
        }
    }

    init(name: String, text: String? = nil) {
        self.name = name
        self.text = text
    }

    deinit {
        print("\(name) 被析构")
    }

}

//创建并打印HTMLElement实例
var paragraph: HTMLElement? = HTMLElement(name: "p", text: "hello,
print(paragraph!.asHTML())

// HTMLElement实例将会被销毁，并能看到它的析构函数打印出的消息
paragraph = nil
```

以上程序执行输出结果为：

```
<p>hello, world</p>
p 被析构
```

Swift 类型转换

Swift 语言类型转换可以判断实例的类型。也可以用于检测实例类型是否属于其父类或者子类的实例。

Swift 中类型转换使用 is 和 as 操作符实现，is 用于检测值的类型，as 用于转换类型。

类型转换也可以用来检查一个类是否实现了某个协议。

定义一个类层次

类型转换用于检测实例类型是否属于特定的实例类型。

你可以将它用在类和子类的层次结构上，检查特定类实例的类型并且转换这个类实例的类型成为这个层次结构中的其他类型。

实例如下：

```
class Subjects {
    var physics: String
    init(physics: String) {
        self.physics = physics
    }
}

class Chemistry: Subjects {
    var equations: String
    init(physics: String, equations: String) {
        self.equations = equations
        super.init(physics: physics)
    }
}

class Maths: Subjects {
    var formulae: String
    init(physics: String, formulae: String) {
        self.formulae = formulae
        super.init(physics: physics)
    }
}

let sa = [
    Chemistry(physics: "固体物理", equations: "赫兹"),
    Maths(physics: "流体动力学", formulae: "千兆赫")]

let samplechem = Chemistry(physics: "固体物理", equations: "赫兹")
print("实例物理学是: \(samplechem.physics)")
print("实例方程式: \(samplechem.equations)")

let samplemaths = Maths(physics: "流体动力学", formulae: "千兆赫")
print("实例物理学是: \(samplemaths.physics)")
print("实例公式是: \(samplemaths.formulae)")
```

以上程序执行输出结果为：

```
实例物理学是: 固体物理
实例方程式: 赫兹
实例物理学是: 流体动力学
实例公式是: 千兆赫
```

检查类型

类型检查使用 **is** 关键字。

操作符 **is** 来检查一个实例是否属于特定子类型。若实例属于那个子类型，类型检查操作符返回 true，否则返回 false。

```
class Subjects {
    var physics: String
    init(physics: String) {
        self.physics = physics
    }
}

class Chemistry: Subjects {
    var equations: String
    init(physics: String, equations: String) {
        self.equations = equations
        super.init(physics: physics)
    }
}

class Maths: Subjects {
    var formulae: String
    init(physics: String, formulae: String) {
        self.formulae = formulae
        super.init(physics: physics)
    }
}

let sa = [
    Chemistry(physics: "固体物理", equations: "赫兹"),
    Maths(physics: "流体动力学", formulae: "千兆赫"),
    Chemistry(physics: "热物理学", equations: "分贝"),
    Maths(physics: "天体物理学", formulae: "兆赫"),
    Maths(physics: "微分方程", formulae: "余弦级数")]

let samplechem = Chemistry(physics: "固体物理", equations: "赫兹")
print("实例物理学是: \(samplechem.physics)")
print("实例方程式: \(samplechem.equations)")

let samplemaths = Maths(physics: "流体动力学", formulae: "千兆赫")
print("实例物理学是: \(samplemaths.physics)")
print("实例公式是: \(samplemaths.formulae)")

var chemCount = 0
var mathsCount = 0
for item in sa {
    // 如果是一个 Chemistry 类型的实例，返回 true，相反返回 false。
    if item is Chemistry {
        ++chemCount
    } else if item is Maths {
        ++mathsCount
    }
}

print("化学科目包含 \(chemCount) 个主题，数学包含 \(mathsCount) 个主题")
```

以上程序执行输出结果为：

```
实例物理学是: 固体物理
实例方程式: 赫兹
实例物理学是: 流体动力学
实例公式是: 千兆赫
化学科目包含 2 个主题，数学包含 3 个主题
```

向下转型

向下转型，用类型转换操作符(as? 或 as!)

当你不确定向下转型可以成功时，用类型转换的条件形式(as?)。条件形式的类型转换总是返回一个可选值（optional value），并且若下转是不可能的，可选值将是nil。

只有你可以确定向下转型一定会成功时，才使用强制形式(as!)。当你试图向下转型为一个不正确的类型时，强制形式的类型转换会触发一个运行时错误。

```
class Subjects {
    var physics: String
    init(physics: String) {
        self.physics = physics
    }
}

class Chemistry: Subjects {
    var equations: String
    init(physics: String, equations: String) {
        self.equations = equations
        super.init(physics: physics)
    }
}

class Maths: Subjects {
    var formulae: String
    init(physics: String, formulae: String) {
        self.formulae = formulae
        super.init(physics: physics)
    }
}

let sa = [
    Chemistry(physics: "固体物理", equations: "赫兹"),
    Maths(physics: "流体动力学", formulae: "千兆赫"),
    Chemistry(physics: "热物理学", equations: "分贝"),
    Maths(physics: "天体物理学", formulae: "兆赫"),
    Maths(physics: "微分方程", formulae: "余弦级数")]

let samplechem = Chemistry(physics: "固体物理", equations: "赫兹")
print("实例物理学是: \(samplechem.physics)")
print("实例方程式: \(samplechem.equations)")

let samplemaths = Maths(physics: "流体动力学", formulae: "千兆赫")
print("实例物理学是: \(samplemaths.physics)")
print("实例公式是: \(samplemaths.formulae)")

var chemCount = 0
var mathsCount = 0

for item in sa {
    // 类型转换的条件形式
    if let show = item as? Chemistry {
        print("化学主题是: '\(show.physics)', \(show.equations)")
        // 强制形式
    } else if let example = item as? Maths {
        print("数学主题是: '\(example.physics)',  \(example.formulae
    }
}
```

以上程序执行输出结果为：

```
实例物理学是: 固体物理
实例方程式: 赫兹
实例物理学是: 流体动力学
实例公式是: 千兆赫
化学主题是: '固体物理', 赫兹
数学主题是: '流体动力学',  千兆赫
化学主题是: '热物理学', 分贝
数学主题是: '天体物理学',  兆赫
数学主题是: '微分方程',  余弦级数
```

Any和AnyObject的类型转换

Swift为不确定类型提供了两种特殊类型别名：

- `AnyObject` 可以代表任何class类型的实例。
- `Any` 可以表示任何类型，包括方法类型（function types）。

> 注意：
>
> 只有当你明确的需要它的行为和功能时才使用 `Any` 和 `AnyObject` 。在你的代码里使用你期望的明确的类型总是更好的。

Any 实例

```
class Subjects {
    var physics: String
    init(physics: String) {
        self.physics = physics
    }
}

class Chemistry: Subjects {
    var equations: String
    init(physics: String, equations: String) {
        self.equations = equations
        super.init(physics: physics)
    }
}

class Maths: Subjects {
    var formulae: String
    init(physics: String, formulae: String) {
        self.formulae = formulae
        super.init(physics: physics)
    }
}
```

```
let sa = [
    Chemistry(physics: "固体物理", equations: "赫兹"),
    Maths(physics: "流体动力学", formulae: "千兆赫"),
    Chemistry(physics: "热物理学", equations: "分贝"),
    Maths(physics: "天体物理学", formulae: "兆赫"),
    Maths(physics: "微分方程", formulae: "余弦级数")]

let samplechem = Chemistry(physics: "固体物理", equations: "赫兹")
print("实例物理学是: \(samplechem.physics)")
print("实例方程式: \(samplechem.equations)")

let samplemaths = Maths(physics: "流体动力学", formulae: "千兆赫")
print("实例物理学是: \(samplemaths.physics)")
print("实例公式是: \(samplemaths.formulae)")

var chemCount = 0
var mathsCount = 0

for item in sa {
    // 类型转换的条件形式
    if let show = item as? Chemistry {
        print("化学主题是: '\(show.physics)', \(show.equations)")
        // 强制形式
    } else if let example = item as? Maths {
        print("数学主题是: '\(example.physics)',  \(example.formulae
    }
}

// 可以存储Any类型的数组 exampleany
var exampleany = [Any]()

exampleany.append(12)
exampleany.append(3.14159)
exampleany.append("Any 实例")
exampleany.append(Chemistry(physics: "固体物理", equations: "兆赫"))

for item2 in exampleany {
    switch item2 {
    case let someInt as Int:
        print("整型值为 \(someInt)")
    case let someDouble as Double where someDouble > 0:
        print("Pi 值为 \(someDouble)")
    case let someString as String:
        print("\(someString)")
    case let phy as Chemistry:
        print("主题 '\(phy.physics)', \(phy.equations)")
    default:
        print("None")
    }
}
```

以上程序执行输出结果为：

```
实例物理学是: 固体物理
实例方程式: 赫兹
实例物理学是: 流体动力学
实例公式是: 千兆赫
化学主题是: '固体物理', 赫兹
数学主题是: '流体动力学',  千兆赫
化学主题是: '热物理学', 分贝
数学主题是: '天体物理学',  兆赫
数学主题是: '微分方程',  余弦级数
整型值为 12
Pi 值为 3.14159
Any 实例
主题 '固体物理', 兆赫
```

AnyObject 实例

```
class Subjects {
    var physics: String
    init(physics: String) {
        self.physics = physics
    }
}

class Chemistry: Subjects {
    var equations: String
    init(physics: String, equations: String) {
        self.equations = equations
        super.init(physics: physics)
    }
}

class Maths: Subjects {
    var formulae: String
    init(physics: String, formulae: String) {
        self.formulae = formulae
        super.init(physics: physics)
    }
}

// [AnyObject] 类型的数组
let saprint: [AnyObject] = [
    Chemistry(physics: "固体物理", equations: "赫兹"),
    Maths(physics: "流体动力学", formulae: "千兆赫"),
    Chemistry(physics: "热物理学", equations: "分贝"),
    Maths(physics: "天体物理学", formulae: "兆赫"),
    Maths(physics: "微分方程", formulae: "余弦级数")]

let samplechem = Chemistry(physics: "固体物理", equations: "赫兹")
```

```
print("实例物理学是: \(samplechem.physics)")
print("实例方程式: \(samplechem.equations)")

let samplemaths = Maths(physics: "流体动力学", formulae: "千兆赫")
print("实例物理学是: \(samplemaths.physics)")
print("实例公式是: \(samplemaths.formulae)")

var chemCount = 0
var mathsCount = 0

for item in saprint {
    // 类型转换的条件形式
    if let show = item as? Chemistry {
        print("化学主题是: '\(show.physics)', \(show.equations)")
        // 强制形式
    } else if let example = item as? Maths {
        print("数学主题是: '\(example.physics)',  \(example.formulae
    }
}

var exampleany = [Any]()
exampleany.append(12)
exampleany.append(3.14159)
exampleany.append("Any 实例")
exampleany.append(Chemistry(physics: "固体物理", equations: "兆赫"))

for item2 in exampleany {
    switch item2 {
    case let someInt as Int:
        print("整型值为 \(someInt)")
    case let someDouble as Double where someDouble > 0:
        print("Pi 值为 \(someDouble)")
    case let someString as String:
        print("\(someString)")
    case let phy as Chemistry:
        print("主题 '\(phy.physics)', \(phy.equations)")
    default:
        print("None")
    }
}
```

以上程序执行输出结果为：

```
实例物理学是: 固体物理
实例方程式: 赫兹
实例物理学是: 流体动力学
实例公式是: 千兆赫
化学主题是: '固体物理', 赫兹
数学主题是: '流体动力学',  千兆赫
化学主题是: '热物理学', 分贝
数学主题是: '天体物理学',  兆赫
数学主题是: '微分方程',  余弦级数
整型值为 12
Pi 值为 3.14159
Any 实例
主题 '固体物理', 兆赫
```

在一个switch语句的case中使用强制形式的类型转换操作符（as, 而不是 as?）来检查和转换到一个明确的类型。

Swift 扩展

扩展就是向一个已有的类、结构体或枚举类型添加新功能。

扩展可以对一个类型添加新的功能，但是不能重写已有的功能。

Swift 中的扩展可以：

- 添加计算型属性和计算型静态属性
- 定义实例方法和类型方法
- 提供新的构造器
- 定义下标
- 定义和使用新的嵌套类型
- 使一个已有类型符合某个协议

语法

扩展声明使用关键字 **extension**：

```
extension SomeType  {  // 加到SomeType的新功能写到这里  }
```

一个扩展可以扩展一个已有类型，使其能够适配一个或多个协议，语法格式如下：

```
extension SomeType:  SomeProtocol,  AnotherProctocol  {  // 协议实现
```

计算型属性

扩展可以向已有类型添加计算型实例属性和计算型类型属性。

实例

下面的例子向 Int 类型添加了 5 个计算型实例属性并扩展其功能：

```
extension Int  {  var add:  Int  {return  self  +  100  }  var  sub
```

以上程序执行输出结果为：

```
加法运算后的值：103  减法运算后的值：110  乘法运算后的值：390  除法运算后的值
```

构造器

扩展可以向已有类型添加新的构造器。

这可以让你扩展其它类型，将你自己的定制类型作为构造器参数，或者提供该类型的原始实现中没有包含的额外初始化选项。

扩展可以向类中添加新的便利构造器 init()，但是它们不能向类中添加新的指定构造器或析构函数 deinit() 。

```
struct sum {  var num1 =  100, num2 =  200  }  struct diff {  var n
        _ = y.no1 + y.no2 }  }  let a = sum(num1:  100, num2:  200)
```

以上程序执行输出结果为：

```
mult 模块内  (100,  200) mult 模块内  (200,  100) mult 模块内  (300,
```

方法

扩展可以向已有类型添加新的实例方法和类型方法。

下面的例子向Int类型添加一个名为 topics 的新实例方法：

```
extension Int  { func topics(summation:  ()  ->  ())  {  for _ in
```

以上程序执行输出结果为：

```
扩展模块内  扩展模块内  扩展模块内  扩展模块内  内型转换模块内  内型转换模块内
```

这个 `topics` 方法使用了一个 `() -> ()` 类型的单参数，表明函数没有参数而且没有返回值。

定义该扩展之后，你就可以对任意整数调用 `topics` 方法,实现的功能则是多次执行某任务：

可变实例方法

通过扩展添加的实例方法也可以修改该实例本身。

结构体和枚举类型中修改self或其属性的方法必须将该实例方法标注为mutating，正如来自原始实现的修改方法一样。

实例

下面的例子向 Swift 的 Double 类型添加了一个新的名为 square 的修改方法，来实现一个原始值的平方计算：

```
extension Double  { mutating func square()  {  let pi =  3.1415  se
```

以上程序执行输出结果为：

```
圆的面积为:  34.210935  圆的面积为:  105.68006  圆的面积为:  45464.0707
```

下标

扩展可以向一个已有类型添加新下标。

实例

以下例子向 Swift 内建类型Int添加了一个整型下标。该下标[n]返回十进制数字

```
extension Int  { subscript(var multtable:  Int)  ->  Int  {  var no
```

以上程序执行输出结果为：

```
2  6  5
```

嵌套类型

扩展可以向已有的类、结构体和枚举添加新的嵌套类型：

```
extension Int  {  enum calc {  case add case  sub  case mult case c
```

以上程序执行输出结果为：

```
10  20  30  40  50  50
```

Swift 协议

协议规定了用来实现某一特定功能所必需的方法和属性。

任意能够满足协议要求的类型被称为遵循(conform)这个协议。

类，结构体或枚举类型都可以遵循协议，并提供具体实现来完成协议定义的方法和功能。

语法

协议的语法格式如下：

```
protocol SomeProtocol {
    // 协议内容
}
```

要使类遵循某个协议，需要在类型名称后加上协议名称，中间以冒号:分隔，作为类型定义的一部分。遵循多个协议时，各协议之间用逗号,分隔。

```
struct SomeStructure: FirstProtocol, AnotherProtocol {
    // 结构体内容
}
```

如果类在遵循协议的同时拥有父类，应该将父类名放在协议名之前，以逗号分隔。

```
class SomeClass: SomeSuperClass, FirstProtocol, AnotherProtocol {
    // 类的内容
}
```

对属性的规定

协议用于指定特定的实例属性或类属性，而不用指定是存储型属性或计算型属性。此外还必须指明是只读的还是可读可写的。

协议中的通常用var来声明变量属性，在类型声明后加上{ set get }来表示属性是可读可写的，只读属性则用{ get }来表示。

```
protocol classa {

    var marks: Int { get set }
    var result: Bool { get }

    func attendance() -> String
    func markssecured() -> String

}

protocol classb: classa {

    var present: Bool { get set }
    var subject: String { get set }
    var stname: String { get set }

}

class classc: classb {
    var marks = 96
    let result = true
    var present = false
    var subject = "Swift 协议"
    var stname = "Protocols"

    func attendance() -> String {
        return "The \(stname) has secured 99% attendance"
    }

    func markssecured() -> String {
        return "\(stname) has scored \(marks)"
    }
}

let studdet = classc()
studdet.stname = "Swift"
studdet.marks = 98
studdet.markssecured()

print(studdet.marks)
print(studdet.result)
print(studdet.present)
print(studdet.subject)
print(studdet.stname)
```

以上程序执行输出结果为：

```
98
true
false
Swift 协议
Swift
```

对 Mutating 方法的规定

有时需要在方法中改变它的实例。

例如，值类型(结构体，枚举)的实例方法中，将mutating关键字作为函数的前缀，写在func之前，表示可以在该方法中修改它所属的实例及其实例属性的值。

```
protocol daysofaweek {
    mutating func show()
}

enum days: daysofaweek {
    case sun, mon, tue, wed, thurs, fri, sat
    mutating func show() {
        switch self {
        case sun:
            self = sun
            print("Sunday")
        case mon:
            self = mon
            print("Monday")
        case tue:
            self = tue
            print("Tuesday")
        case wed:
            self = wed
            print("Wednesday")
        case mon:
            self = thurs
            print("Thursday")
        case tue:
            self = fri
            print("Friday")
        case sat:
            self = sat
            print("Saturday")
        default:
            print("NO Such Day")
        }
    }
}

var res = days.wed
res.show()
```

以上程序执行输出结果为：

```
Wednesday
```

对构造器的规定

协议可以要求它的遵循者实现指定的构造器。

你可以像书写普通的构造器那样，在协议的定义里写下构造器的声明，但不需要写花括号和构造器的实体，语法如下：

```
protocol SomeProtocol {
   init(someParameter: Int)
}
```

实例

```
protocol tcpprotocol {
   init(aprot: Int)
}
```

协议构造器规定在类中的实现

你可以在遵循该协议的类中实现构造器，并指定其为类的指定构造器或者便利构造器。在这两种情况下，你都必须给构造器实现标上"required"修饰符：

```
class SomeClass: SomeProtocol {
   required init(someParameter: Int) {
      // 构造器实现
   }
}

protocol tcpprotocol {
   init(aprot: Int)
}

class tcpClass: tcpprotocol {
   required init(aprot: Int) {
   }
}
```

使用required修饰符可以保证：所有的遵循该协议的子类，同样能为构造器规定提供一个显式的实现或继承实现。

如果一个子类重写了父类的指定构造器，并且该构造器遵循了某个协议的规定，那么该构造器的实现需要被同时标示required和override修饰符：

```
protocol tcpprotocol {
    init(no1: Int)
}

class mainClass {
    var no1: Int // 局部变量
    init(no1: Int) {
        self.no1 = no1 // 初始化
    }
}

class subClass: mainClass, tcpprotocol {
    var no2: Int
    init(no1: Int, no2 : Int) {
        self.no2 = no2
        super.init(no1:no1)
    }
    // 因为遵循协议，需要加上"required"; 因为继承自父类，需要加上"override
    required override convenience init(no1: Int)  {
        self.init(no1:no1, no2:0)
    }
}
let res = mainClass(no1: 20)
let show = subClass(no1: 30, no2: 50)

print("res is: \(res.no1)")
print("res is: \(show.no1)")
print("res is: \(show.no2)")
```

以上程序执行输出结果为：

```
res is: 20
res is: 30
res is: 50
```

协议类型

尽管协议本身并不实现任何功能，但是协议可以被当做类型来使用。

协议可以像其他普通类型一样使用，使用场景:

- 作为函数、方法或构造器中的参数类型或返回值类型
- 作为常量、变量或属性的类型
- 作为数组、字典或其他容器中的元素类型

实例

```
protocol Generator {
    typealias members
    func next() -> members?
}

var items = [10,20,30].generate()
while let x = items.next() {
    print(x)
}

for lists in [1,2,3].map( {i in i*5}) {
    print(lists)
}

print([100,200,300])
print([1,2,3].map({i in i*10}))
```

以上程序执行输出结果为：

```
10
20
30
5
10
15
[100, 200, 300]
[10, 20, 30]
```

在扩展中添加协议成员

我们可以可以通过扩展来扩充已存在类型(类，结构体，枚举等)。

扩展可以为已存在的类型添加属性，方法，下标脚本，协议等成员。

```
protocol AgeClasificationProtocol {
   var age: Int { get }
   func agetype() -> String
}

class Person {
   let firstname: String
   let lastname: String
   var age: Int
   init(firstname: String, lastname: String) {
      self.firstname = firstname
      self.lastname = lastname
      self.age = 10
   }
}

extension Person : AgeClasificationProtocol {
   func fullname() -> String {
      var c: String
      c = firstname + " " + lastname
      return c
   }

   func agetype() -> String {
      switch age {
      case 0...2:
         return "Baby"
      case 2...12:
         return "Child"
      case 13...19:
         return "Teenager"
      case let x where x > 65:
         return "Elderly"
      default:
         return "Normal"
      }
   }
}
```

协议的继承

协议能够继承一个或多个其他协议，可以在继承的协议基础上增加新的内容要求。

```
protocol InheritingProtocol: SomeProtocol, AnotherProtocol {
    // 协议定义
}
```

实例

```
protocol Classa {
    var no1: Int { get set }
    func calc(sum: Int)
}

protocol Result {
    func print(target: Classa)
}

class Student2: Result {
    func print(target: Classa) {
        target.calc(1)
    }
}

class Classb: Result {
    func print(target: Classa) {
        target.calc(5)
    }
}

class Student: Classa {
    var no1: Int = 10

    func calc(sum: Int) {
        no1 -= sum
        print("学生尝试 \(sum) 次通过")

        if no1 <= 0 {
            print("学生缺席考试")
        }
    }
}

class Player {
    var stmark: Result!

    init(stmark: Result) {
        self.stmark = stmark
    }

    func print(target: Classa) {
        stmark.print(target)
    }
}

var marks = Player(stmark: Student2())
var marksec = Student()

marks.print(marksec)
```

```
marks.print(marksec)
marks.print(marksec)
marks.stmark = Classb()
marks.print(marksec)
marks.print(marksec)
marks.print(marksec)
```

以上程序执行输出结果为：

```
学生尝试 1 次通过
学生尝试 1 次通过
学生尝试 1 次通过
学生尝试 5 次通过
学生尝试 5 次通过
学生缺席考试
学生尝试 5 次通过
学生缺席考试
```

类专属协议

你可以在协议的继承列表中,通过添加class关键字,限制协议只能适配到类（class）类型。

该class关键字必须是第一个出现在协议的继承列表中，其后，才是其他继承协议。格式如下：

```
protocol SomeClassOnlyProtocol: class, SomeInheritedProtocol {
    // 协议定义
}
```

实例

```
protocol TcpProtocol {
    init(no1: Int)
}

class MainClass {
    var no1: Int // 局部变量
    init(no1: Int) {
        self.no1 = no1 // 初始化
    }
}

class SubClass: MainClass, TcpProtocol {
    var no2: Int
    init(no1: Int, no2 : Int) {
        self.no2 = no2
        super.init(no1:no1)
    }
    // 因为遵循协议，需要加上"required"; 因为继承自父类，需要加上"override
    required override convenience init(no1: Int)  {
        self.init(no1:no1, no2:0)
    }
}

let res = MainClass(no1: 20)
let show = SubClass(no1: 30, no2: 50)

print("res is: \(res.no1)")
print("res is: \(show.no1)")
print("res is: \(show.no2)")
```

以上程序执行输出结果为：

```
res is: 20
res is: 30
res is: 50
```

协议合成

Swift 支持合成多个协议，这在我们需要同时遵循多个协议时非常有用。

语法格式如下：

```
protocol<SomeProtocol, AnotherProtocol>
```

实例

```
protocol Stname {
    var name: String { get }
}

protocol Stage {
    var age: Int { get }
}

struct Person: Stname, Stage {
    var name: String
    var age: Int
}

func show(celebrator: protocol<Stname, Stage>) {
    print("\(celebrator.name) is \(celebrator.age) years old")
}

let studname = Person(name: "Priya", age: 21)
print(studname)

let stud = Person(name: "Rehan", age: 29)
print(stud)

let student = Person(name: "Roshan", age: 19)
print(student)
```

以上程序执行输出结果为：

```
Person(name: "Priya", age: 21)
Person(name: "Rehan", age: 29)
Person(name: "Roshan", age: 19)
```

检验协议的一致性

你可以使用is和as操作符来检查是否遵循某一协议或强制转化为某一类型。

- `is` 操作符用来检查实例是否 `遵循` 了某个 `协议` 。
- `as?` 返回一个可选值，当实例 `遵循` 协议时，返回该协议类型;否则返回 `nil` 。
- `as` 用以强制向下转型，如果强转失败，会引起运行时错误。

实例

下面的例子定义了一个 HasArea 的协议，要求有一个Double类型可读的 area：

```
protocol HasArea {
    var area: Double { get }
}

// 定义了Circle类，都遵循了HasArea协议
class Circle: HasArea {
    let pi = 3.1415927
    var radius: Double
    var area: Double { return pi * radius * radius }
    init(radius: Double) { self.radius = radius }
}

// 定义了Country类，都遵循了HasArea协议
class Country: HasArea {
    var area: Double
    init(area: Double) { self.area = area }
}

// Animal是一个没有实现HasArea协议的类
class Animal {
    var legs: Int
    init(legs: Int) { self.legs = legs }
}

let objects: [AnyObject] = [
    Circle(radius: 2.0),
    Country(area: 243_610),
    Animal(legs: 4)
]

for object in objects {
    // 对迭代出的每一个元素进行检查，看它是否遵循了HasArea协议
    if let objectWithArea = object as? HasArea {
        print("面积为 \(objectWithArea.area)")
    } else {
        print("没有面积")
    }
}
```

以上程序执行输出结果为：

```
面积为 12.5663708
面积为 243610.0
没有面积
```

Swift 泛型

Swift 提供了泛型让你写出灵活且可重用的函数和类型。

Swift 标准库是通过泛型代码构建出来的。

Swift 的数组和字典类型都是泛型集。

你可以创建一个Int数组，也可创建一个String数组，或者甚至于可以是任何其他 Swift 的类型数据数组。

以下实例是一个非泛型函数 exchange 用来交换两个 Int 值：

```
// 定义一个交换两个变量的函数
func exchange(inout a: Int, inout b: Int) {
    let temp = a
    a = b
    b = temp
}

var numb1 = 100
var numb2 = 200

print("交换前数据: \(numb1) 和 \(numb2)")
exchange(&numb1, b: &numb2)
print("交换后数据: \(numb1) 和 \(numb2)")
```

以上程序执行输出结果为：

```
交换前数据: 100 和 200
交换后数据: 200 和 100
```

泛型函数可以访问任何类型，如 Int 或 String。

以下实例是一个泛型函数 exchange 用来交换两个 Int 和 String 值：

```
func exchange<T>(inout a: T, inout b: T) {
    let temp = a
    a = b
    b = temp
}

var numb1 = 100
var numb2 = 200

print("交换前数据:  \(numb1) 和 \(numb2)")
exchange(&numb1, b: &numb2)
print("交换后数据: \(numb1) 和 \(numb2)")

var str1 = "A"
var str2 = "B"

print("交换前数据:  \(str1) 和 \(str2)")
exchange(&str1, b: &str2)
print("交换后数据: \(str1) 和 \(str2)")
```

以上程序执行输出结果为：

```
交换前数据:  100 和 200
交换后数据: 200 和 100
交换前数据:  A 和 B
交换后数据: B 和 A
```

这个函数的泛型版本使用了占位类型名字（通常此情况下用字母T来表示）来代替实际类型名（如Int、String或Double）。占位类型名没有提示T必须是什么类型，但是它提示了a和b必须是同一类型T，而不管T表示什么类型。只有 exchange(::)函数在每次调用时所传入的实际类型才能决定T所代表的类型。

另外一个不同之处在于这个泛型函数名后面跟着的占位类型名字（T）是用尖括号括起来的（<t>）。这个尖括号告诉 Swift 那个T是 exchange(::)函数所定义的一个类型。因为T是一个占位命名类型，Swift 不会去查找命名为T的实际类型。</t>

泛型类型

Swift 允许你定义你自己的泛型类型。

自定义类、结构体和枚举作用于任何类型，如同Array和Dictionary的用法。

```
struct TOS<T> {
    var items = [T]()
    mutating func push(item: T) {
        items.append(item)
    }

    mutating func pop() -> T {
        return items.removeLast()
    }
}

var tos = TOS<String>()
tos.push("Swift")
print(tos.items)

tos.push("泛型")
print(tos.items)

tos.push("类型参数")
print(tos.items)

tos.push("类型参数名")
print(tos.items)

let deletetos = tos.pop()
```

以上程序执行输出结果为：

```
["Swift"]
["Swift", "泛型"]
["Swift", "泛型", "类型参数"]
["Swift", "泛型", "类型参数", "类型参数名"]
```

扩展泛型类型

当你扩展一个泛型类型的时候（使用 extension 关键字），你并不需要在扩展的定义中提供类型参数列表。更加方便的是，原始类型定义中声明的类型参数列表在扩展里是可以使用的，并且这些来自原始类型中的参数名称会被用作原始定义中类型参数的引用。

实例

```
struct TOS<T> {
    var items = [T]()
    mutating func push(item: T) {
        items.append(item)
    }

    mutating func pop() -> T {
        return items.removeLast()
    }
}

var tos = TOS<String>()
tos.push("Swift")
print(tos.items)

tos.push("泛型")
    print(tos.items)

    tos.push("类型参数")
    print(tos.items)

    tos.push("类型参数名")
    print(tos.items)

// 扩展泛型 TOS 类型
extension TOS {
    var first: T? {
        return items.isEmpty ? nil : items[items.count - 1]
    }
}

if let first = tos.first {
    print("栈顶部项：\(first)")
}
```

以上程序执行输出结果为：

```
["Swift"]
["Swift", "泛型"]
["Swift", "泛型", "类型参数"]
["Swift", "泛型", "类型参数", "类型参数名"]
栈顶部项：类型参数名
```

类型约束

类型约束指定了一个必须继承自指定类的类型参数，或者遵循一个特定的协议或协议构成。

类型约束语法

你可以写一个在一个类型参数名后面的类型约束，通过冒号分割，来作为类型参数链的一部分。这种作用于泛型函数的类型约束的基础语法如下所示（和泛型类型的语法相同）：

```
func someFunction<T: SomeClass, U: SomeProtocol>(someT: T, someU: U
    // 这里是函数主体
}
```

实例

```
// 函数可以作用于查找一字符串数组中的某个字符串
func findStringIndex(array: [String], _ valueToFind: String) -> Int
    for (index, value) in array.enumerate() {
        if value == valueToFind {
            return index
        }
    }
    return nil
}

let strings = ["cat", "dog", "llama", "parakeet", "terrapin"]
if let foundIndex = findStringIndex(strings, "llama") {
    print("llama 的下标索引值为 \(foundIndex)")
}
```

以上程序执行输出结果为：

```
llama 的下标索引值为 2
```

关联类型实例

Swift 中使用 typealias 关键字来设置关联类型。

定义一个协议时，有的时候声明一个或多个关联类型作为协议定义的一部分是非常有用的。

```
protocol Container {
    // 定义了一个ItemType关联类型
    typealias ItemType
    mutating func append(item: ItemType)
    var count: Int { get }
    subscript(i: Int) -> ItemType { get }
}

// 遵循Container协议的泛型TOS类型
struct TOS<T>: Container {
    // original Stack<T> implementation
    var items = [T]()
    mutating func push(item: T) {
        items.append(item)
    }

    mutating func pop() -> T {
        return items.removeLast()
    }

    // conformance to the Container protocol
    mutating func append(item: T) {
        self.push(item)
    }

    var count: Int {
        return items.count
    }

    subscript(i: Int) -> T {
        return items[i]
    }
}

var tos = TOS<String>()
tos.push("Swift")
print(tos.items)

tos.push("泛型")
print(tos.items)

tos.push("参数类型")
print(tos.items)

tos.push("类型参数名")
print(tos.items)
```

以上程序执行输出结果为：

```
["Swift"]
["Swift", "泛型"]
["Swift", "泛型", "参数类型"]
["Swift", "泛型", "参数类型", "类型参数名"]
```

Where 语句

类型约束能够确保类型符合泛型函数或类的定义约束。

你可以在参数列表中通过where语句定义参数的约束。

你可以写一个where语句，紧跟在在类型参数列表后面，where语句后跟一个或者多个针对关联类型的约束，以及（或）一个或多个类型和关联类型间的等价(equality)关系。

实例

下面的例子定义了一个名为allItemsMatch的泛型函数，用来检查两个Container实例是否包含相同顺序的相同元素。

如果所有的元素能够匹配，那么返回一个为true的Boolean值，反之则为false。

```
protocol Container {
    typealias ItemType
    mutating func append(item: ItemType)
    var count: Int { get }
    subscript(i: Int) -> ItemType { get }
}

struct Stack<T>: Container {
    // original Stack<T> implementation
    var items = [T]()
    mutating func push(item: T) {
        items.append(item)
    }

    mutating func pop() -> T {
        return items.removeLast()
    }

    // conformance to the Container protocol
    mutating func append(item: T) {
        self.push(item)
    }

    var count: Int {
        return items.count
    }
```

```
    subscript(i: Int) -> T {
        return items[i]
    }
}

func allItemsMatch<
    C1: Container, C2: Container
    where C1.ItemType == C2.ItemType, C1.ItemType: Equatable>
    (someContainer: C1, anotherContainer: C2) -> Bool {
        // 检查两个Container的元素个数是否相同
        if someContainer.count != anotherContainer.count {
            return false
        }

        // 检查两个Container相应位置的元素彼此是否相等
        for i in 0..<someContainer.count {
            if someContainer[i] != anotherContainer[i] {
                return false
            }
        }
        // 匹配所有项，返回 true
        return true
}

var tos = Stack<String>()
tos.push("Swift")
print(tos.items)

tos.push("泛型")
print(tos.items)

tos.push("Where 语句")
print(tos.items)

var eos = ["Swift", "泛型", "Where 语句"]
print(eos)
```

以上程序执行输出结果为：

```
["Swift"]
["Swift", "泛型"]
["Swift", "泛型", "Where 语句"]
["Swift", "泛型", "Where 语句"]
```

Swift 访问控制

访问控制可以限定其他源文件或模块中代码对你代码的访问级别。

你可以明确地给单个类型（类、结构体、枚举）设置访问级别，也可以给这些类型的属性、函数、初始化方法、基本类型、下标索引等设置访问级别。

协议也可以被限定在一定的范围内使用，包括协议里的全局常量、变量和函数。

访问控制基于模块与源文件。

模块指的是以独立单元构建和发布的Framework或Application。在Swift 中的一个模块可以使用import关键字引入另外一个模块。

源文件是单个源码文件，它通常属于一个模块， 源文件可以包含多个类和函数 的定义。

Swift 为代码中的实体提供了三种不同的访问级别:public、internal、private。

访问级别	定义
Public	可以访问自己模块中源文件里的任何实体，别人也可以通过引入该模块来访问源文件里的所有实体。
Internal	：可以访问自己模块中源文件里的任何实体，但是别人不能访问该模块中源文件里的实体。
Private	只能在当前源文件中使用的实体，称为私有实体。

public为最高级访问级别，private为最低级访问级别。

语法

通过修饰符public、internal、private来声明实体的访问级别：

```
public class SomePublicClass {}
internal class SomeInternalClass {}
private class SomePrivateClass {}

public var somePublicVariable = 0
internal let someInternalConstant = 0
private func somePrivateFunction() {}
```

除非有特殊的说明，否则实体都使用默认的访问级别internal。

函数类型访问权限

函数的访问级别需要根据该函数的参数类型和返回类型的访问级别得出。

下面的例子定义了一个名为someFunction全局函数，并且没有明确地申明其访问级别。

```
func someFunction() -> (SomeInternalClass, SomePrivateClass) {
    // 函数实现
}
```

函数中其中一个类 SomeInternalClass 的访问级别是internal，另一个 SomePrivateClass 的访问级别是private。所以根据元组访问级别的原则，该元组的访问级别是private。

```
private func someFunction() -> (SomeInternalClass, SomePrivateClass
    // 函数实现
}
```

将该函数申明为public或internal，或者使用默认的访问级别internal都是错误的。

枚举类型访问权限

枚举中成员的访问级别继承自该枚举，你不能为枚举中的成员单独申明不同的访问级别。

实例

比如下面的例子，枚举 Student 被明确的申明为 public 级别，那么它的成员 Name，Mark 的访问级别同样也是 public：

```
public enum Student {
    case Name(String)
    case Mark(Int,Int,Int)
}

var studDetails = Student.Name("Swift")
var studMarks = Student.Mark(98,97,95)

switch studMarks {
case .Name(let studName):
    print("学生名: \(studName).")
case .Mark(let Mark1, let Mark2, let Mark3):
    print("学生成绩: \(Mark1),\(Mark2),\(Mark3)")
}
```

以上程序执行输出结果为：

```
学生成绩: 98,97,95
```

子类访问权限

子类的访问级别不得高于父类的访问级别。比如说，父类的访问级别是internal，子类的访问级别就不能申明为public。

```
public class SuperClass {
    private func show() {
        print("超类")
    }
}

// 访问级别不能低于超类 internal > public
internal class SubClass: SuperClass  {
    override internal func show() {
        print("子类")
    }
}

let sup = SuperClass()
sup.show()

let sub = SubClass()
sub.show()
```

以上程序执行输出结果为：

```
超类
子类
```

常量、变量、属性、下标访问权限

常量、变量、属性不能拥有比它们的类型更高的访问级别。

比如说，你定义一个public级别的属性，但是它的类型是private级别的，这是编译器所不允许的。

同样，下标也不能拥有比索引类型或返回类型更高的访问级别。

如果常量、变量、属性、下标索引的定义类型是private级别的，那么它们必须要明确的申明访问级别为private:

```
private var privateInstance = SomePrivateClass()
```

Getter 和 Setter访问权限

常量、变量、属性、下标索引的Getters和Setters的访问级别继承自它们所属成员的访问级别。

Setter的访问级别可以低于对应的Getter的访问级别，这样就可以控制变量、属性或下标索引的读写权限。

```
class Samplepgm {
    private var counter: Int = 0{
        willSet(newTotal){
            print("计数器: \(newTotal)")
        }
        didSet{
            if counter > oldValue {
                print("新增加数量 \(counter - oldValue)")
            }
        }
    }
}

let NewCounter = Samplepgm()
NewCounter.counter = 100
NewCounter.counter = 800
```

以上程序执行输出结果为：

```
计数器: 100
新增加数量 100
计数器: 800
新增加数量 700
```

构造器和默认构造器访问权限

初始化

我们可以给自定义的初始化方法申明访问级别，但是要不高于它所属类的访问级别。但必要构造器例外，它的访问级别必须和所属类的访问级别相同。

如同函数或方法参数，初始化方法参数的访问级别也不能低于初始化方法的访问级别。

默认初始化方法

Swift为结构体、类都提供了一个默认的无参初始化方法，用于给它们的所有属性提供赋值操作，但不会给出具体值。

默认初始化方法的访问级别与所属类型的访问级别相同。

实例

在每个子类的 init() 方法前使用 required 关键字声明访问权限。

```
class classA {
    required init() {
        var a = 10
        print(a)
    }
}

class classB: classA {
    required init() {
        var b = 30
        print(b)
    }
}

let res = classA()
let show = classB()
```

以上程序执行输出结果为：

```
10
30
10
```

协议访问权限

如果想为一个协议明确的申明访问级别，那么需要注意一点，就是你要确保该协议只在你申明的访问级别作用域中使用。

如果你定义了一个public访问级别的协议，那么实现该协议提供的必要函数也会是public的访问级别。这一点不同于其他类型，比如，public访问级别的其他类型，他们成员的访问级别为internal。

```
public protocol TcpProtocol {
    init(no1: Int)
}

public class MainClass {
    var no1: Int // local storage
    init(no1: Int) {
        self.no1 = no1 // initialization
    }
}

class SubClass: MainClass, TcpProtocol {
    var no2: Int
    init(no1: Int, no2 : Int) {
        self.no2 = no2
        super.init(no1:no1)
    }

    // Requires only one parameter for convenient method
    required override convenience init(no1: Int)  {
        self.init(no1:no1, no2:0)
    }
}

let res = MainClass(no1: 20)
let show = SubClass(no1: 30, no2: 50)

print("res is: \(res.no1)")
print("res is: \(show.no1)")
print("res is: \(show.no2)")
```

以上程序执行输出结果为：

```
res is: 20
res is: 30
res is: 50
```

扩展访问权限

你可以在条件允许的情况下对类、结构体、枚举进行扩展。扩展成员应该具有和原始类成员一致的访问级别。比如你扩展了一个公共类型，那么你新加的成员应该具有和原始成员一样的默认的internal访问级别。

或者，你可以明确申明扩展的访问级别（比如使用private extension）给该扩展内所有成员申明一个新的默认访问级别。这个新的默认访问级别仍然可以被单独成员所申明的访问级别所覆盖。

泛型访问权限

泛型类型或泛型函数的访问级别取泛型类型、函数本身、泛型类型参数三者中的最低访问级别。

```
public struct TOS<T> {
    var items = [T]()
    private mutating func push(item: T) {
        items.append(item)
    }

    mutating func pop() -> T {
        return items.removeLast()
    }
}

var tos = TOS<String>()
tos.push("Swift")
print(tos.items)

tos.push("泛型")
print(tos.items)

tos.push("类型参数")
print(tos.items)

tos.push("类型参数名")
print(tos.items)
let deletetos = tos.pop()
```

以上程序执行输出结果为：

```
["Swift"]
["Swift", "泛型"]
["Swift", "泛型", "类型参数"]
["Swift", "泛型", "类型参数", "类型参数名"]
```

类型别名

任何你定义的类型别名都会被当作不同的类型，以便于进行访问控制。一个类型别名的访问级别不可高于原类型的访问级别。

比如说，一个private级别的类型别名可以设定给一个public、internal、private的类型，但是一个public级别的类型别名只能设定给一个public级别的类型，不能设定给internal或private 级别的类型。

> 注意： 这条规则也适用于为满足协议一致性而给相关类型命名别名的情况。

```
public protocol Container {
    typealias ItemType
    mutating func append(item: ItemType)
    var count: Int { get }
    subscript(i: Int) -> ItemType { get }
}

struct Stack<T>: Container {
    // original Stack<T> implementation
    var items = [T]()
    mutating func push(item: T) {
        items.append(item)
    }

    mutating func pop() -> T {
        return items.removeLast()
    }

    // conformance to the Container protocol
    mutating func append(item: T) {
        self.push(item)
    }

    var count: Int {
        return items.count
    }

    subscript(i: Int) -> T {
        return items[i]
    }
}

func allItemsMatch<
```

```
    C1: Container, C2: Container
    where C1.ItemType == C2.ItemType, C1.ItemType: Equatable>
    (someContainer: C1, anotherContainer: C2) -> Bool {
        // check that both containers contain the same number of it
        if someContainer.count != anotherContainer.count {
            return false
        }

        // check each pair of items to see if they are equivalent
        for i in 0..<someContainer.count {
            if someContainer[i] != anotherContainer[i] {
                return false
            }
        }

        // all items match, so return true
        return true
}

var tos = Stack<String>()
tos.push("Swift")
print(tos.items)

tos.push("泛型")
print(tos.items)

tos.push("Where 语句")
print(tos.items)

var eos = ["Swift", "泛型", "Where 语句"]
print(eos)
```

以上程序执行输出结果为：

```
["Swift"]
["Swift", "泛型"]
["Swift", "泛型", "Where 语句"]
["Swift", "泛型", "Where 语句"]
```